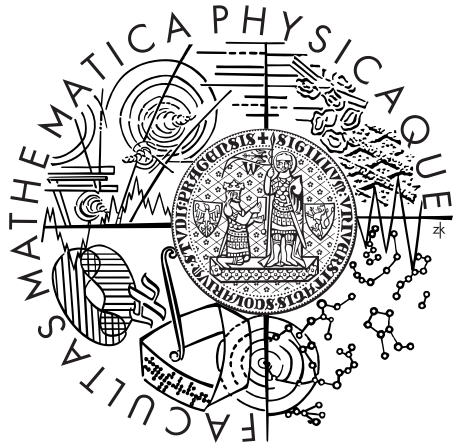
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Ondřej David

Identifying runtime components of the running SOFA 2 application

Department of Software Engineering

Supervisor: RNDr. Tomáš Bureš, Ph.D.

Study Program: Computer Science, Software Systems

I would like to express my thanks to my supervisor Tomáš Bureš for his guidance on my thesis. I appreciate his time spent on explaining me the SOFA 2 runtime environment, his numerous suggestions about improving the thesis and his constant willingness to give me pieces of advice every time I needed them.

I hereby declare that I have elaborated my master thesis on my own and listed all used references. I agree with making the thesis publicly available.

Prague, August 3, 2009 Ondřej David

Contents

Název práce: Identifying runtime components of the running SOFA 2 application
Autor: Ondřej David
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí diplomové práce: RNDr. Tomáš Bureš, Ph.D.
e-mail vedoucího: bures@dsrg.mff.cuni.cz

Abstrakt: Tato práce se zabývá návrhem dekompozice komponentového běhového prostředí SOFA 2. První část je věnovaná popisu logických částí prostředí a jejich vzájemných vazeb. Jsou zde popsány úlohy podpůrných programových celků (repository, dock manager, global connector manager, dock) nezbytných ke spuštění libovolné komponentové architektury, která se skládá z konkrétních běhových části (jádro komponenty, mikrokomponenty, konektory) popsaných dále v textu. Druhá část hledá odpověď na otázku jak konkrétní komponentová architektura ovlivňuje přítomnost těchto běhových části. Jejich skladbu, vazby mezi sebou a závislosti zkoumá na základě pozorování konkrétních komponentových aplikací. Společné rysy zachycuje návrhem sjednocujícího konceptu, díky kterému bude možné zjednodušit a odlehčit výsledné běhové prostředí konkrétních komponentových architektur. Třetí část ověřuje proveditelnost navrhovaného konceptu. Pomocí OSGi bundlů jsou zde implementována běhová prostředí komponentových aplikací zkoumaných ve druhé části textu a na implementaci se demonstruje správnost navrhovaného konceptu. Závěrečná část shrnuje dosažené výsledky a nabízí možnosti, jak lze na práci dál pokračovat.

Klíčová slova: SOFA 2, běhové prostředí, komponenty

Title: Identifying runtime components of the running SOFA 2 application
Author: Ondřej David
Department: Department of Software Engineering
Supervisor: RNDr. Tomáš Bureš, Ph.D.
Supervisor's e-mail address: bures@dsrg.mff.cuni.cz

Abstract: In the present work we put forward a proposal on how to deconstruct a SOFA 2 component runtime environment. The first part of the text is dedicated to description of its logical parts and their interconnections. It includes description of the subsidiary program parts (repository, dock manager, global connector manager, dock) essential for launching an arbitrary component architecture which binds additional runtime components (component content, microcomponents, connectors) described further on in the text. The second part deals with the question of how a specific component architecture influences presence of these runtime components. Their constitution, bindings and dependencies are investigated based on observations of various component applications. Common traits are captured in a unifying concept which will open the door to making the target runtime environment for concrete component architecture more configurable and lightweight. The third part of the text proves the feasibility of the proposed concept. OSGi bundles are used to implement the runtime environments of the component applications investigated in the second part of the text demonstrating correctness of the proposed concept. The final part summarises the results of the text and presents possible ways how to continue in this work.

Keywords: SOFA 2, runtime environment, components

Chapter 1

Introduction

Components are for composition. Composition enables premade "things" to be reused by rearranging them in brand new composites. In the context of computer science the word component referes to a *software componet* (henceforth refered to as the component) which [Szyperski] defines as an executable unit of independent production, acquisition and deployment that can be composed into a functioning system. The definition casts several criteria on a software entity it needs to fulfill in order that it could be called a component:

- Multiple-use i.e., it should be possible to plug the component into various functioning systems
- Non-context-specific i.e., the component should be unaware of the surrounding environment into which it is plugged in
- Composable with other components i.e., the component should be able to communicate with the components it is connected with
- Encapsulated i.e., the component should be non-investigable through its interfaces

1.1 Why components

From a broad point of view the software development strategies have two poles. On one side there is the extreme programming strategy where the projects are developed from a scratch. These projects build only on libraries shipped together with the programming language. On the other side there is the strategy of composing own applications exclusively from a third party software which needs to be configurable enough to meet the user's needs. Both strategies have their pros and cons discussed in the next two paragraphs.

As far as the first pole of extreme programming is considered, it brings along one big advantage. It can be optimized to fit in the user's business model and take advantage of the user's proprietary solutions. This fact pulls the solution ahead of the competition which offers general solutions. On the other hand it is bug prone which is why it costs a lot of resources to keep the solution stable and working properly. The implementation of these projects is consequently time-consuming and in the world of rapid change they usually become obsolete before productive. For big projects such development strategy poses a substantial risk to be imlemented and realized.

The opposite pole of having an application made of exclusively third party software bears considerable benefits. If the application is wholy outsourced the risks can be limited by fixed prices captured in contracts. If the application is constituted of standard software it is not necessary to take care of the maintainance and evolution of these parts since it is guaranteed by the vendor. It is only necessary to configure the software correctly and ensure proper migration if a new version of the software is released. One of the flaws of this strategy is obvious - hardly ever optimal solution. The standard software is always general and needs to be parametrized which never brings optimal solution. Moreover the user is usually forced to change their internal business processes to fit the standard solutions. At last if the competition uses the same software it will never be possible to get ahead of it.

The component based development [Szyperski] is trying to ease problems of both strategies and keep their advantages at the same time. If the solution is cut into logical parts on each level of abstraction, it is possible to implement each logical part with a component. Each component can be then custom made, outsourced or realized by means of a standard software and thus grasp advantages of each individual development strategy and limit their shortcomings. If a fixed budget is given, for each component it is necessary to consider how good level of performance it should have, its resource efficiency, robustness, degree of certification, etc. in order to find as much optimal solution as possible.

1.2 Component systems

Every project has several phases of development: requirements elicitation, analysis of the problem, architecture, etc.. During the architecture phase it is being discussed how to realize the proposed solution from the analysis phase, and that is the moment when the component based development comes into play. The output of the architecture phase should be a *component architecture* which introduces a set of components each having well defined provided and required interfaces and definition of the connections between these components. Once all the components are implemented and tested, it is time to pour life into the component architecture. For this purpose a *component system* is introduced. Component system is a set of subsidiary tools which understands the given component architecture and knows how to instantiate it. Every component system has its *component model* which is a metamodel of component architectures that can be instantiated by the system. It is important that the component system is chosen before the component architecture is designed because every system and its component model can provide various advanced features that some component architectures might require. The component systems are generally divided into academic systems and systems driven by industry. The academic systems put emphasis on advanced features at the expense of lacking strong *runtime environment* which could hold and run the advanced architectures. On the other hand the systems driven by industry usually have simple metamodel as they concentrate on the runtime environment and its robustness. The next paragraph will cover several such systems and describe their features.

1.2.1 Academic systems

Major part of currently used component systems stems from the historical system Darwin [5]. It is a system that completely lacks runtime environment part and only introduces a standard way of how to define a component architecture using ADL [4]. It provides hierarchical

component architectures without connectors (the connection between components is not a first-class concept).

Like Darwin, Wright [6] presents a new ADL which formalizes a software architecture in terms of concepts such as components, connectors, roles, and ports. The dynamic behavior of different ports of an individual component and its roles are described using the Communicating Sequential Processes (CSP) process algebra. Due to the formal nature of the behavior descriptions, automatic checks of port/role compatibility and overall system consistency can be performed.

1.2.2 Systems driven by industry

Unlike systems mentioned in the previous paragraph, EJB [7] is a full-featured component runtime environment designed for the use in commercial fields. However, it does not provide as many features as for example the Wright environment. It provides only flat component architectures, method invocation and messages as far as the communication styles are concerned, yet no support for formal verification of the composition and no support for distribution.

Koala [8] is a component system with runtime environment support designed for embedded systems[1]. From the list of advanced features it only allows hierarchical components since the embedded systems cannot hold heavy loaded applications. Koala uses a special designed compiler which generates a C code where the implementation of components has to be filled in. For the sake of optimalization it can perform partial evaluation, thus find unreachable code or components and remove them.

1.2.3 SOFA 2

The component system this thesis concentrates on, is the full-featured component runtime environment SOFA 2 [2, 3]. It allows for hierarchically composed components, it provides ADL-based design, behavior specification using behavior protocols, automatically generated connectors at runtime supporting seamless and transparent distribution of applications, aspect-based management logic, etc. It serves as a distributed runtime environment with dynamic update of components. It is an academic project continuously being developed by the Distributed Systems Research Group at Charles University in Prague. In comparison with other academic environments it provides a full-featured runtime environment with support for versioning, repository where the components reside, and other features usually found in environments driven by industry. SOFA 2 compared to these environments supports hierarchical component architectures, seamless distribution of components, multiple communication styles, runtime modification of the component architecture and behavior verifications.

[1] A special-purpose computer that performs a few dedicated functions, usually with very specific requirements. (e.g. portable devices such as cell phones or MP3 players, flight control systems, car engine controllers, home appliances, medical equipment, TV sets, etc.)

1.3 Motivation

SOFA 2 suffers from several shortcomings. It behaves like a monolith. It encapsulates several mutually heterogeneous concepts (ADL-based design, connector architecture, hierarchically nested components, etc.) which makes SOFA 2 difficult to be used in variuos IT domains.

One of these domains are embedded systems. Their common characteristic is their lightweight nature. They are limited by their low computational power and low main memory size in comparison with desktop computers. These constraints make it difficult to port SOFA 2 in the embedded systems. Let us take for example a pocket MP3 player. It has few functional blocks: access to the simple file system, MP3 file decoder and access to the output device (headphones). These functional needs could be met by a simple component architecture constitued of three components interconnected by a stream connector. Despite the simple architecture, the SOFA runtime will have to go through all the procedures (i.e. XML parsing, application of aspect-based management logic, automatical connector generation, etc.) necessary for instantiating the architecture. Due to the hardware limitations of the player these procedures can never be carried out on the device.

Another domain are the long running or mission critical systems, especially systems which evolve during execution. Without an ability of evolving during the lifetime, the high costs and risks associated with shutting down and restarting these systems would have to be considered. Many such systems employ elaborate mechanisms that allow system extension during execution. The modification of runtime business components in such systems can be tightly coupled with the modification of the runtime environment part. Let us consider a group of client components connected via a connector to a group of server components each running on a different computer. The connector serves as a load balancer. The number of servers can change during the time. It can temporarily drop due to maintainance purposes of any of the servers or it can grow because another server component has been added to the group. In all cases it is necessary to at least notify the connector that the number of servers has changed. The design of SOFA 2 is not currently able to address this runtime requirement. The ADL-based design of component architectures strictly determines the constitution of the runtime which is therefore static and unflexible.

1.4 Problem statement

The overall design of the SOFA 2 runtime environment is monolithic and static. It is memory consuming and thus impossible to be used in embedded systems due to their hardware limitations. It is impossible to handle (modify, replace, communicate with) a specific part of the runtime environment during the application lifetime.

1.5 Goals

The goal of the thesis is to find a unifying concept proposing the way how to deconstruct the SOFA 2 runtime environment. The concept should underlie all the mutually heterogeneous concepts currently found in SOFA 2 by identifying their runtime parts depending on a specific component application. As soon as the parts are identified and their dependencies on component architectures are investigated, it will be possible to make a transparent and fully modular runtime environment. Such environment will be able to meet the limitations of

lightweight systems and the requirements introduced by the long running or mission critical systems, especially systems which evolve during execution.

1.6 The structure of the thesis

The structure of the thesis is as follows. The following chapter 2 describes the SOFA 2 runtime environment and its concepts focusing on their runtime parts in running SOFA 2 applications. Chapter 3 proposes a metamodel unifying the runtime parts in one concept hiding the overall heterogenity. Chapter 4 proves the proposed concept to be correct and feasible by demonstrating its application on several SOFA 2 applications. Chapter 5 discusses related work dealing with similar issue of transforming sophisticated concept into a simple concept. The chapter 6 concludes and summarizes contribution of the work.

Chapter 2

SOFA 2

SOFA 2 is a component system employing several advanced concepts. It introduces components as encapsulated entities each of which declares a set of provided and required interfaces through which they communicate with each other.

The SOFA 2 component model distinguishes between two types of components. A component can be either primitive or composite. A primitive component has a content which is the business code implementing the component's behaviour. The content of a composite component is another set of interconnected subcomponents. The composite component delegates business interfaces calls onto its subcomponents.

Every component has a control part separated from the business part. It mediates and realizes non-functional features of the component such as access to the list of provided/required/control interfaces, logging of business interfaces calls, management of the component's lifecycle, etc.. These features are managed by the aspect-based logic which makes it easily extensible.

A connection between two interfaces is realized by a first-class concept, the connector. It is realized as a hyper-edge which means any number of components can take part in the connection. The concept enables seamless distribution of the components, it can reflect any kind of communication paradigm (RPC, asynchronous messaging, streaming) or apply any security policy.

SOFA also supports a runtime evolution. It offers a factory evolution pattern which can be followed in order to create a new component and attach it to a component's interface during the application runtime.

2.1 Defining a component architecture

Every SOFA 2 component architecture is constituted of several business entites defined in ADL [4]. The first step in building a component architecture is the declaration of all *business interface*s consequently used by the business components whose definition follows. The definition of business components is accomplished in two substeps. The first one is the declaration of *frame*s which only publish set of required and provided business interfaces together with their communication styles. The next substep is the declaration of *architecture*s which declare what frames they implement and are real representatives of the final business components. If an architecture represents a composite business component, it contains list of its business subcomponents (further on referred to as the subcomponents) along with a list of

connections between these subcomponents. The following paragraphs capture all business entities in more detail.

2.1.1 Business interface

The business interface entity is an independent element through which the business components communicate. Its declaration takes one parameter referencing the signature of the java interface which defines the interface on the level of the programming language.

2.1.2 Frame

A frame is a black-box view of a component. It gives a list of provided and required business interfaces. The frame also states the way of communication for each interface it involves. The communication style options are *method-invocation*, *streaming* and *messaging*. The detailed description of particular communication styles is elaborated in section 2.5. A frame does not have any connection with the programming language level as it only represents a point of view at a business component.

2.1.3 Architecture

An architecture is a grey-box view of a business component. Every architecture implements one or more frames and inherits their provided and required business interfaces. If an architecture represents a primitive business component, it contains reference on its Java business implementation, the *content*. Otherwise the architecture represents a composite business component and it refers its subcomponents as another frames or architectures. Since a subcomponent can be another architecture, it is possible to create as many levels of nesting as needed. An architecture of a composite component also specifies the interconnections between its subcomponents and the connections between its interfaces and the subcomponents. Every component architecture has its root architecture which encompasses all other top-level components and specifies connections between them.

2.2 Runtime environment

The runtime environment of SOFA 2 is constituted of several subsidiary parts which altogether take part in instantiation of a specific component architecture. They are refered to as the *SOFA node*. The SOFA node comprises a *repository*, a *deployment dock registry*, a *global connector manager* and a set of *deployment dock*s. Next few paragraphs will refer to them in more detail.

2.2.1 Repository

Repository is a persistent storage for storing all information needed for instantiating a component architecture. It stores implementation of particular business components as well as implementation of its runtime parts such as types of connections (method invocation, streaming, messaging), security policy, runtime factory evolution pattern or delegation of calls in the composite components. Furthermore it contains the *deployment plan* which serves as a guide on how to instantiate the architecture.

2.2.2 Deployment dock

A deployment dock is a container where components are launched. It provides necessary infrastructure for managing the components' lifecycle. Every component forming an application is assigned to a specific dock in the SOFA node according to the deployment plan. The tool managing the deployment reads the deployment plan and instructs all the docks involved to download the components' implementation from the repository and to instantiate them.

2.2.3 Deployment dock registry

The deployment dock registry registers all docks involved in a single SOFA 2 node. It is used by the the tool managing the instantiation of a component architecture to lookup nodes mentioned in the deployment plan. Each SOFA 2 node has a single instance of the deployment dock registry.

2.2.4 Global connector manager

The global connector manager is responsible for linking connector units (see 2.5) of connectors together. Each SOFA 2 node has a single instance of the global connector manager.

2.3 Introduction to the levels of abstraction

An instance of a component architecture can be viewed from various levels of abstraction. The top point of view employes several advanced concepts needed to fulfill the business requirements and divides the architecture into two basic parts. The first, the business part, describes the architecture as a set of composite or primitive components connected through well defined interfaces by means of connectors of custom type with an ability of creating new business components at runtime. The second, the control part, is defined as a set of aspects which manage the components' control features. Each aspect may specify a new *control interface* added to the component and a set of supplementary *microcomponents* implementing the control interfaces' logic and extending the component's non-functional abilities.

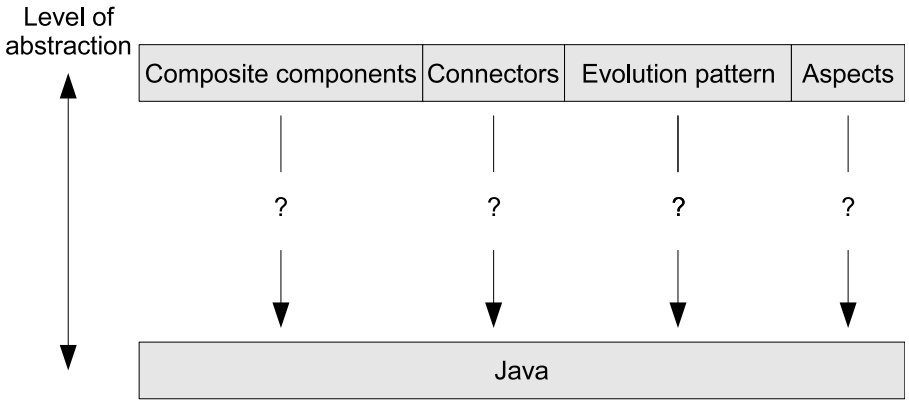


Figure 2.1: Levels of abstraction

The space between the top level perspective and the next underlying level of abstraction spans a remarkable empty gap (figure 2.1) missing a concept capturing the architecture elements on a lower level of abstraction. The nearest underlying level of abstraction is very low - a set of plain java objects each having just set of attributes and methods.

Next sections will deal will the description of the advanced concepts SOFA 2 employs and chapter 3 will put forward a proposal of a concept filling the gap between the two abstraction levels and unifying all advanced concepts in one.

2.4 Control part

The control part of a business component is strictly separated from its business part. It is formed by a set of control interfaces and by a set of microcomponents which are independent logical units both managed by *aspect*s. A control interface is an entry point to the business component's control part.

The set of microcomopnents forming a business component's control part can be viewed as a component architecture which does not employ any of the advanced features. They have a set of provided and required interfaces whose definition is given by their signature in Java, they do not have any hierarchical nesting of the microcomponents, the connection between microcomponents is not a first-class concept (from the point of view of a microcomponent the connection is realized by referencing the other microcomponent) and the architecture is static, in other words no evolution patterns are applied. Microcomponents can be used in two different roles. The first role is a *delegate microcomponent* which stands for a microcomponent associated with one of the the business component's interfaces. These microcomponents are responsible for processing calls going through these interfaces. The second role is just a standalone microcomponent communicating with the other microcomponents.

The delegate microcomponents must provide and require so called delegation interface which has the same signature as the business component's interface whose calls they process. The delegate microcomponents associated with one interface are organized in a chain-like formation, the *delegation chain*, connected through the delegation interface (figure 2.2). For each business component's interface there exists exactly one delegation chain processing the interface's calls. Once a call is performed on that interface, it is delegated to the first microcomponent in the chain. The call is processed by that microcomponent and forwarded to the second microcomponent in the chain and so forth. If the delegation chain stands for a business provided interface, the last microcomponent in the chain hands the call over to the component's content. If the interface is required, the call starts in the content and at the end of the delegation chain it is handed over to the connector (section 2.5). The calls going through the delegation chain representing a control interface end in the last microcomponent of the chain.

The microcomponents used for procession of business interface calls are generated at runtime since the definitions of the interfaces are not known in advance.

2.4.1 Aspects

An aspect is a XML-based tool for management of the control part of a business component. Every aspect at first optionally declares control interfaces which have to be added to the component. After that it defines a set of microcomponents forming the control part. Each

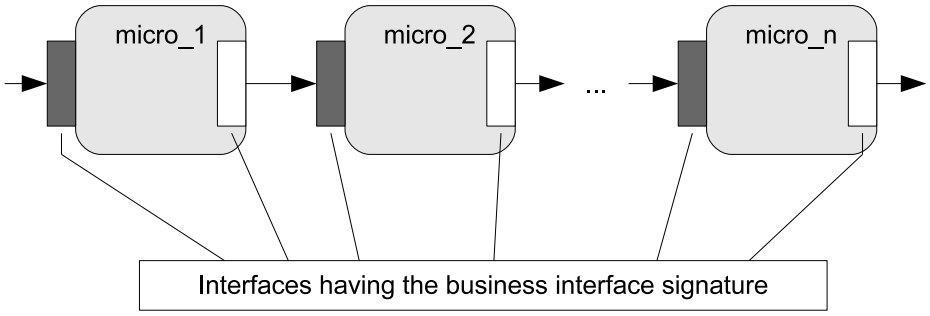


Figure 2.2: Logical structure of a business interface

microcomponent is declared either as a standalone microcomponent or it is associated with any of the component's interface to which it will serve as a delegate. The aspect eventually declares connections between the microcomponents.

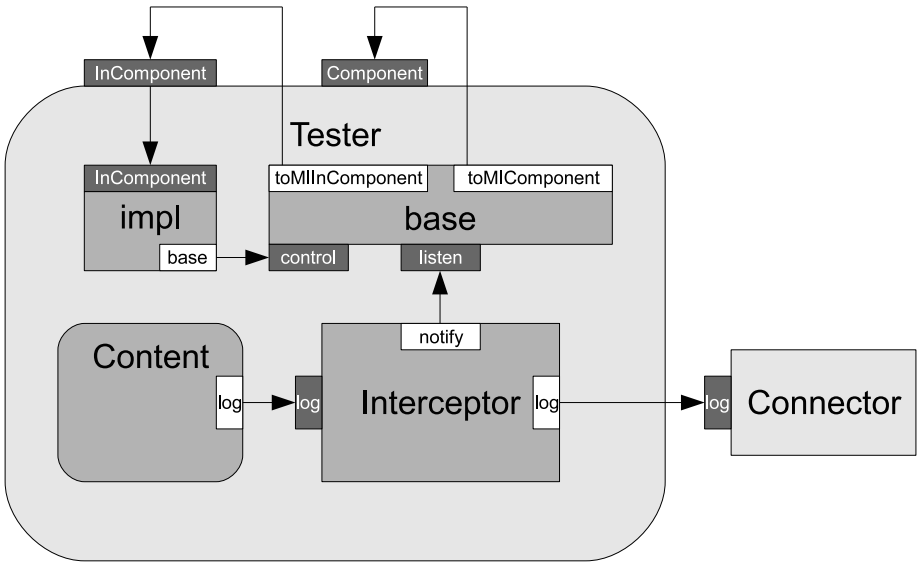


Figure 2.3: Application of an aspect on the `Tester` component

Figure 2.3 demonstrates application of an aspect identified as `InComponent` on a business component named `Tester`. The `Tester` requires one business interface, the `log`. The purpose of the aspect is to provide the component with the ability of keeping the track of threads calling the business interfaces and additionally grant access to the control part of that component through the `Component` control interface. The functionality is realized through microcomponents denoted as `impl`, `base` and `interceptor`. The `impl` microcomponent constitutes a delegation chain for the `InComponent` control interface. The `base` microcomponent is a standalone microcomponent and the `interceptor` microcomponent forms the delegation chain for the business required interface `log`. The microcomponents referencing the control interfaces in reality reference the first microcomponents in their delegation chains.

Once the business content calls any function of the log interface, the call is caught in the `interceptor` which notifies the `base` microcomponent that a call is being performed.

The `base` microcomponent associates the currently running thread with the reference on the `InComponent` interface. The call is returned back to the `interceptor` microcomponent and si delegated forth to the `log` connector. Any code running henceforth under the same thread will have access to the `InComponent` interface. The `InComponent` interface provides function for retrieving reference on the `Component` interface of the component. The calling of this function will be delegated through the `impl` microcomponent to the `base` microcomponent which created the association between the running thread and the `InComponent` interface which implies the reference on the `Component` interface it returns will be of the component where the business call was performed.

2.5 Connectors

A *connector* is a first-class concept which mediates connection between business interfaces of the same type. SOFA 2 offers three types of connections - method invocation, stream connection and message connection. This section will uncover general rules the connector structure obeys and in more detail will examine particular connection types.

A general notion about a connector is that it creates a connection between two components, which means the $1 : 1$ cardinality. SOFA 2, however, offers connections that can be of the $n : n$ cardinality. A simple example of such connection is the well-known client/server paradigm with optional number of clients and also optional number of servers.

A building block of a connector is the *connector element*. It is a primitive entity with internal logic and with a set of provided and required interfaces. The connector elements can be hierarchically nested and the calls are delegated to their subelements. The nesting and the delegation of the calls is not, however, a first-class concept. On the level of a programming language the nested elements are class members of their parent element. The parent element delegates the call by taking one of its members and invoking appropriate function on it.

Each business interface is associated with one connector element. Such element is called the *connector unit*. A set of connector units associated with interfaces involved in one connection form a connector of the connection. The bindings between the corresponding connector units are accomplished through the global connector manager (subsection 2.2.4) which manages the connectors and their connector units.

The primary task of the connector elements is to propagate the business calls through their provided/required interfaces having the same signature as the business interfaces whose calls need to be propagated. The rest of their interfaces is for the purposes of managing connections among the connector units and their connections with the delegation chains. The following list contains such interfaces and gives a brief description to each of them.

- `ElementLocalServer` interface retrieves connector element which is capable of propagating business interface calls further on through the connector.[1]
- `ElementRemoteServer` interface returns information needed for locating the connector element.[2]
- `ElementRemoteClient` interface is the entry point through which the global connector manager provides the connector element with the location of connector elements it

[1] Used by the mechanism which connects delegation chains and the connector units.
[2] Used by the global connector manager which manages connections between the connector units.

needs to connect to.[2]

- `ElementLocalClient` interface is for providing the connector element with the delegation chain into which it should propagate the business calls.[3]

A connector unit is not usually the element which implements the business interfaces and propagates their calls. It contains subelements which do this job instead and the connector unit only mediates these subelements to the delegation chain which propagates the calls from the component's content through its microcomponents to one of these connector elements.

2.5.1 Communication styles

This subsection will deal with the structure of connectors for particular communication styles. Without loss of generality there will be considered only connections of the $1 : 1$ cardinality.

Method invocation connector

The method invocation connector (figure 2.4) has on each side of the connection a connector unit with one subelement. The implementation of the business interface have the subelements which propagate the business calls through the connector. The connector units only mediate their subelements to the delegation chains of their respective business interfaces. The subelement on the client side references directly the subelement on the server side and invokes the propriate function on it when delegating the business call. If the two components being connected are not present in the same dock, the call is realized through RMI [9].

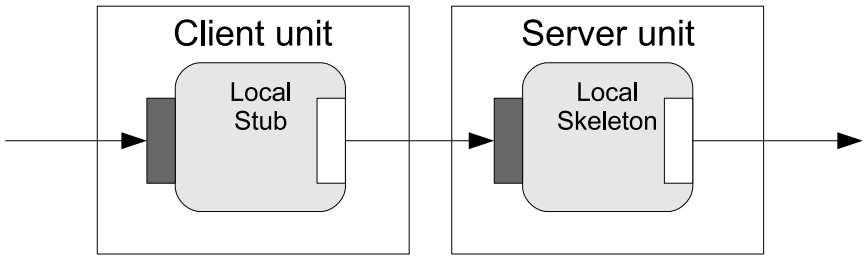


Figure 2.4: Method invocation connector

Stream connector

The stream connector (figure 2.5) does not use any wrapping connector elements. On each side of the connection there is one connector unit which implements the propriate business interface. In comparison with the method invocation connector the client unit does not reference the server unit directly. The server unit listens on given port and the global connector manager delivers the port number along with the IP address of the server unit to the client. The client then communicates with the server through a socket. The business interface serves for retrieving the communicaton stream.

[3]Used by the mechanism which connects connector units with delegation chains.

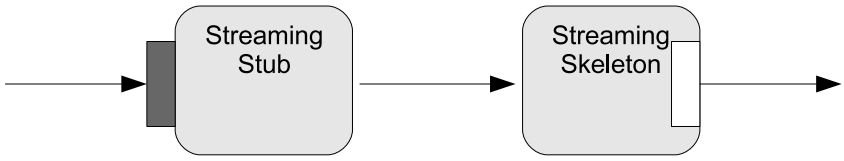


Figure 2.5: Stream connector

Message connector

The message connector (figure 2.6) uses three connector units. Two of them are associated with their respective business interfaces and the third, the `distribution unit`, serves as a broker of the messages (it gathers the messages from the publisher and broadcasts them to the subscriber). The two connector units both contain two subelements - the `adaptor` and the `send receive` unit. The role of the `Send/Receive adaptor` is the transformation of the message from/to the form acceptable by the business interfaces to/from the serializable form. The `send receive` unit takes care of sending/receiving the message to/from the `distribution unit`. The connection between the `send receive unit` and the `distribution unit` is realized through a socket without any direct or remote reference.

The business interfaces the message connector connects does not need to have the same signature. The component sending messages requires interface with function for sending a message. The component receiving messages provides interface with function for receiving the message. The message connector uses these functions to deliver message from one point of communication to another.

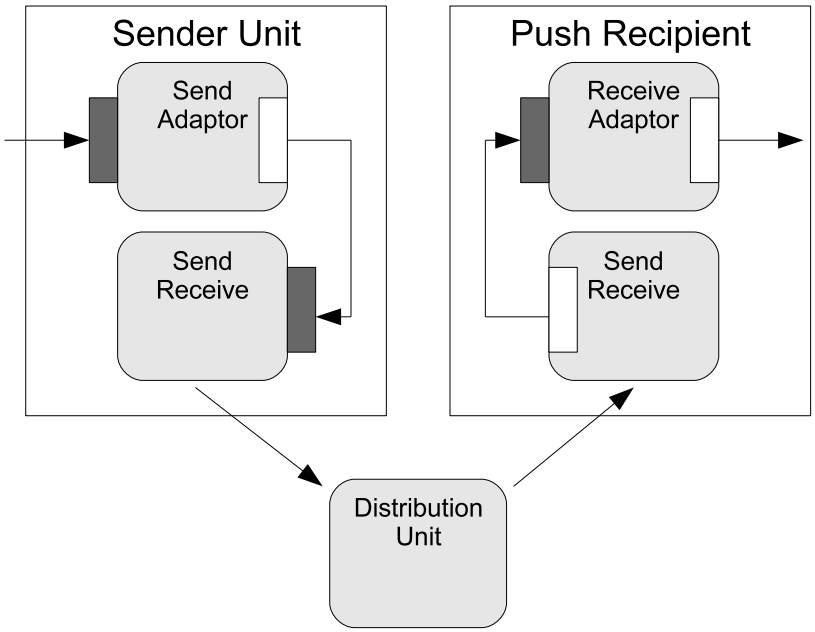


Figure 2.6: Message connector

2.6 Composite components

The composition means nesting a business component inside another business component and connecting the business interfaces of the outer component with the business interfaces of the nested component. As described in section 2.4 each interface is associated with a delegation chain of microcomponents processing and propagating its calls. The section 2.5 described internal structure of connectors of various types. The concept of nesting components exploits the delegation chains and connectors to fulfill its purpose.

The concept operates three logical structures. The delegation chain of the outer business interface, the connector capable of propagating the business interface's calls and the delegation chain of the nested business interface. The two delegation chains and the connector have one parameter in common - the signature of the business interfaces whose calls they process. As they share this signature it is possible to connect the last microcomponent of the outer delegation chain with the connector element provided by the connector as its entry point. In the same manner connect the connector leaving point with the nested delegation chain (figure 2.7).

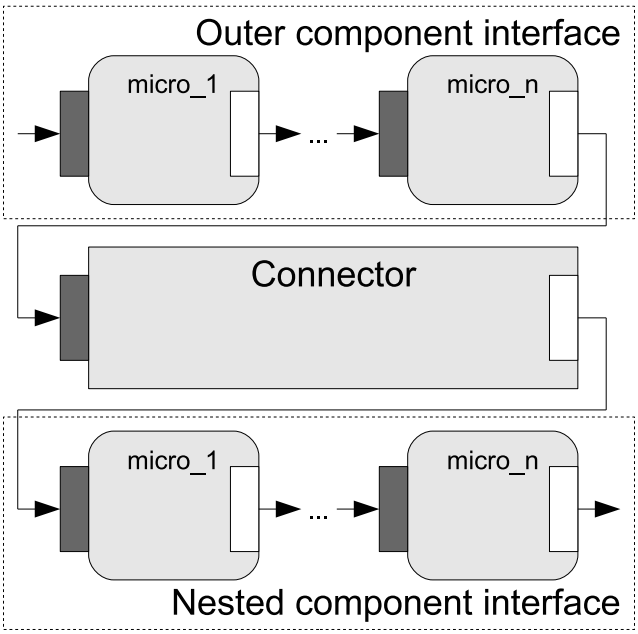


Figure 2.7: Connection between a component and its subcomponent

2.7 Factory evolution pattern

The factory evolution pattern is used for creation of new business components. The creator of such components is a standard business component providing a *factory interface* which is on the business definition level annotated with three parameters. The first parameter specifies the name of the *factory function* in the interface which creates the new component. The second parameter says what frame the new component implements and the third parameter determines which one of the business interfaces, the new component provides, is the return value of the factory function.

The business component implementing the factory interface is responsible for creation of the new component. Its business content must be aware of the component's internals so that it could construct its new instance and return it. The business component which receives the new component breaks the rule of strict separation of the control and business part of a component. Its business content which requires the factory interface is assumed to have a collection of objects annotated as `@Required`. Once the control part receives the newly created component, it injects the instance into this collection and then hands it over to the component's business content.

In terms of the internal structure of the factory evolution pattern, the only remarkable part is inside the business component which asks for a new component. The delegation chain connecting the component's business content with the connector leading to the factory component contains an additional microcomponent, the `Factory Interceptor`, which is responsible for handling the new component. As soon as it is received, the microcomponent creates a new delegation chain whose endpoint is attached to the delegation chain representing the business interface of the new component. The delegation chain's entry point is injected (figure 2.8) to the collection of required business interfaces managed by the business content to which the entry point is handed over afterwards.

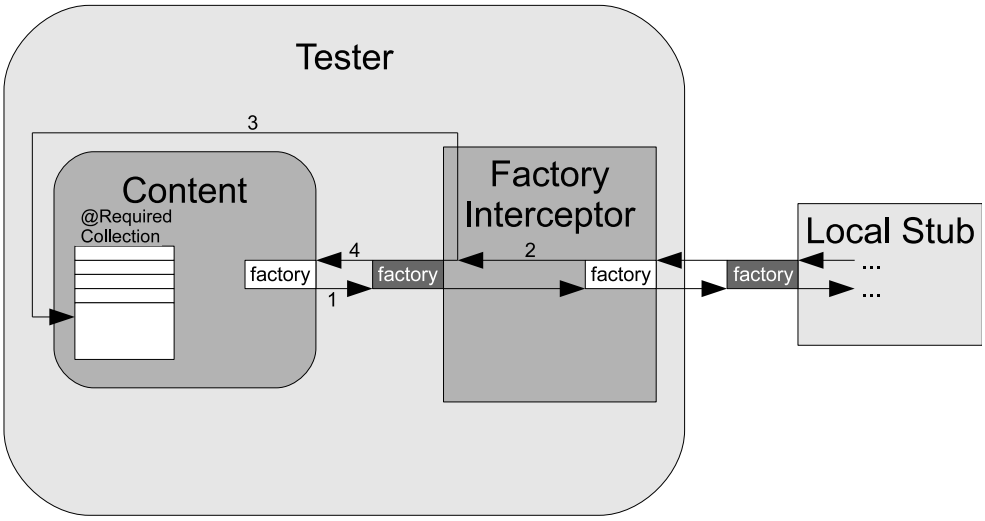


Figure 2.8: New component injection

Chapter 3

Unifying concept

The previous chapter gave an insight into SOFA 2 advanced concepts and described their runtime entities. The top abstraction level of these heterogenous concepts through which a component architecture is modelled is suitable for designers who only need to specify the business part of the architecture and who optionally want to add some non-functional features to the business components without taking care of the implementation details of the runtime environment. Unfortunately, the higher level of abstraction is employed in the design of any software, the higher demands on the computational power are casted on the computer system which would run the software. This is a problem for lightweight systems which could not process a component architecture defined in the top abstraction level. This chapter will put forward a proposal of a unifying concept capturing a component architecture on a level of abstraction between the top level and the level of a programming language. The unifying concept is based on an idea of modelling a component architecture along with all advanced concepts using a simple component architecture which does not employ any of the advanced features (figure 3.1).

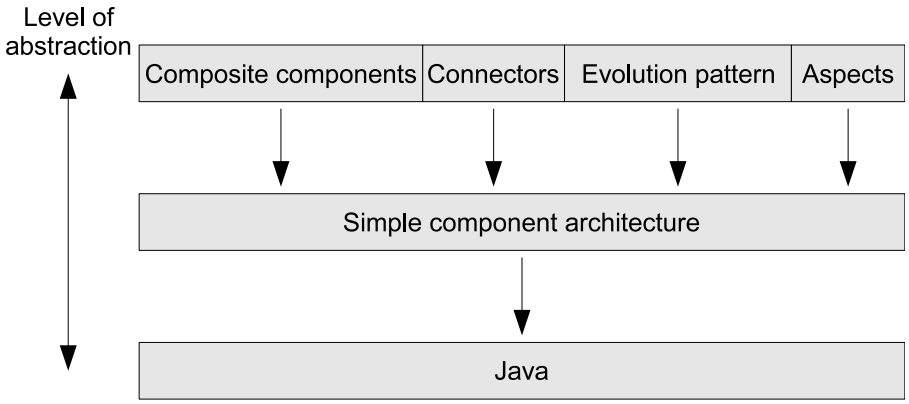


Figure 3.1: Unifying concept abstraction level

3.1 Simple component architecture

A simple component architecture consists of four types of objects as portrays its metamodel in figure 3.2. The basic type is the *simple component* which represents a simple logical unit

and behaves like a black box. Part of every simple component is a set of provided and required *port*s each of which is associated with a specific *interface* type. Ports serve for declaration of connections between two simple components. These connections are represented by the object type *connection*. One connection involves two ports from two different simple components. These ports are not involved in any other connection. These ports must be associated with the same interface type. One of these ports is member of the set of required ports and the other is member of the set of provided ports of the component they belong to.

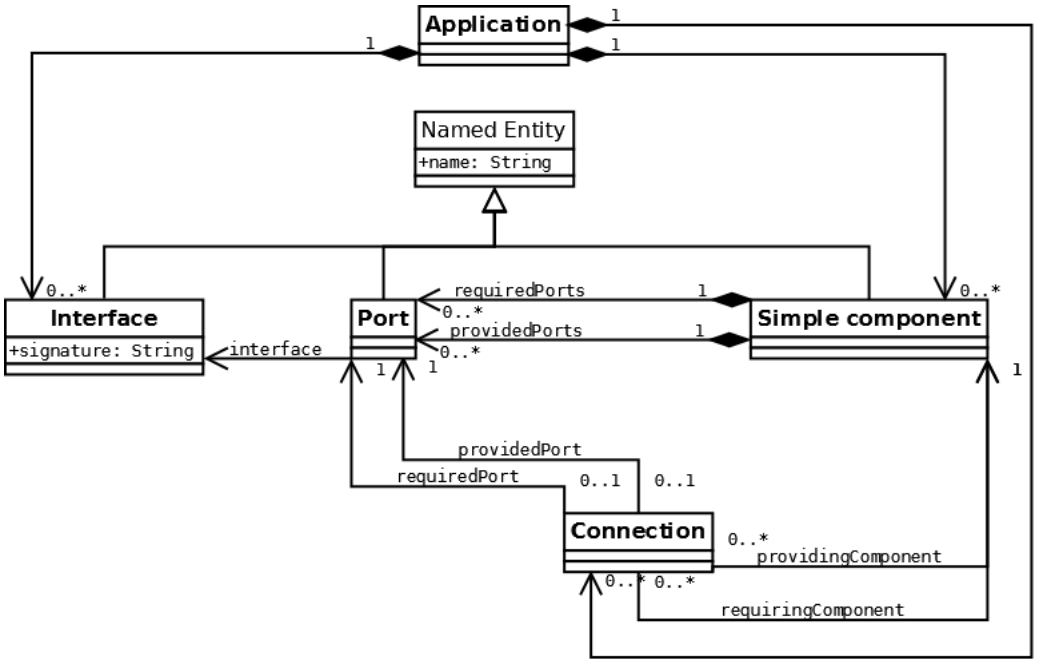


Figure 3.2: Unifying concept metamodel

Constraints of the unifying concept metamodel:

```
context Connection:
  self.providingComponent->providedPorts->
    includes(self.providedPort);
  self.requiringComponent->requiredPorts->
    includes(self.requiredPort);
  self.providingComponent <> self.requiringComponent
  self.providedPort->interface = self.requiredPort->interface

context Application:
  self.connections->forAll(
    c1, c2 : Connection |
      (c1.providedPort = c2.providedPort
      or
      c1.requiredPort = c2.requiredPort)
        implies c1 = c2
  )
```

The following paragraphs will transform every concept found in SOFA 2 into a simple component architecture. Every logical entity found in particular concept will be represented by a *simple component prototype* (henceforth referred to as the prototype). Instances of the logical entity will be then represented by instances of their respective prototype. The internal implementation of the prototype will be identical to the one of the logical entity. The interconnections between the logical entities will be transformed into connections between the prototypes.

Once all the concepts are unified in one, it will be possible to model any SOFA 2 component architecture as a simple component architecture. The whole runtime environment will become transparent, modular and more flexible. It will be possible to take the whole simple component architecture and reconfigure it with respect to the target environment where the application should be run. This will be an asset for lightweight systems which will be relieved from the heavy-loaded SOFA 2 runtime environment.

3.2 Template for transformation description

Each section describing a transformation of a group of logical elements into a simple component architecture will contain two parts:

Description

This section will give a brief description of the elements' role in the SOFA 2 component architecture.

Transformation

This section will give four types of tables describing the simle component architecture onto which the logical elements will be transformed.

Table 3.1: List of simple component prototypes

SOFA 2 entity	Prototype name
The SOFA 2 entity the prototype represents	Simple component prototype name

Table 3.2: Provided ports by prototype X

Port name	Signature
name of the port	signature of the interface the port is associated with

Table 3.3: Required ports by prototype X

Port name	Signature
name of the port	signature of the interface the port is associated with

Table 3.4: List of connections

Requiring prototype	Required port	Providing prototype	Provided port
name of the prototype requiring a port	name of the required port	name of the prototype providing a port	name of the provided port

The first table (3.1) gives list of prototypes in the simple component architecture. For each prototype there will be a pair of tables (3.2, 3.3) giving a list of its required and provided ports. Table 3.4 gives list of connections among the prototypes listed in table 3.1. In case any of these tables would be left empty it will be omitted in the listings.

3.3 Basic concept

The first concept which needs to be transformed is the basic concept to which all the advanced concepts adhere. It covers a business component itself and its business implementation, the content.

3.3.1 Business component content

Description

The business component content is the business implementation of the component and is supplied by the third party which means it is a monolith. It is supposed to provide and require interfaces defined by the business component's architecture.

Table 3.5: Simple component prototypes

SOFA 2 entity	Prototype name
business component content	`core`

Table 3.6: Provided ports by `core`

Port name	Signature
"names of all business provided interfaces by the content"	"signatures of all respective business provided interfaces by the content"

Table 3.7: Required ports by `core`

Port name	Signature
"names of all business required interfaces by the content"	"signatures of all respective business required interfaces by the content"

Transformations

The business component content will be represented by one simple component `core`. It will require and provide ports whose names and signatures correspond with the business interfaces the business content requires and provides.

3.3.2 Business component

Description

The business component itself does not have any internal logic and serves only as a holder of entities it is constituted of. It comprises a business content and sets of delegation chains representing provided, required and control interfaces.

Transformations

Table 3.8: Simple component prototypes

SOFA 2 entity	Prototype name
Business component	`componentInstance`

Table 3.9: Provided ports by `componentInstance`

Port name	Signature
`management`	`org.objectweb.dsrg.sofa.microarchitecture.ComponentInstance`

The `componentInstance` provides one port for management of the components it is constituted of and does not require any port.

3.4 Control part

The control part of a business component is managed by aspects. It supplies the component architecture with two logical entities - microcomponents and the delegation chains.

3.4.1 Delegation chain

Description

A delegation chain is a representative of a business component interface. It is constituted of a chain of microcomponents which process and propagate calls addressed to the interface.

Transformations

Table 3.10: Simple component prototypes

SOFA 2 entity	Prototype name
Delegation chain	`delgChain`

Table 3.11: Provided ports by `delgChain`

Port name	Signature
`management`	`org.objectweb.dsrg.sofa.microarchitecture.DelegationChain`

The `delgChain` provides one port for management of the delegation chain. It contains functions for inserting new microcomponents into it and for retrieving the first microcomponent in the chain. The `delgChain` does not require any port.

3.4.2 Microcomponent

Description

A microcomponent is a building block of the business component control part. It connects the component's content with the world outside the component. It also implements the component's non-functional features.

Transformations

Table 3.12: Simple component prototypes

SOFA 2 entity	Prototype name
Microcomponent	`mc`

Table 3.13: Provided ports by `mc`

Port name	Signature
`management`	`org.objectweb.dsrg.sofa.microarchitecture.SOFAMicroComponent`
`delgProvided`	"signature of the business component interface whose calls the microcomponent delegates".
"names of all other interfaces required by the microcomponent"	"signatures of all respective interfaces required by the microcomponent"

Table 3.14: Required ports by `mc`

Port name	Signature
`delgRequired`	"signature of the business component interface whose calls the microcomponent delegates".
"names of all other interfaces provided by the microcomponent"	"signatures of all respective interfaces provided by the microcomponent"

The `delgProvided` and `delgRequired` ports are provided/required only when the microcomponent is a delegate. In case the microcomponent is the last member of a delegation chain representing a control interface, the port `delgRequired` is not required.

3.5 Connectors

A connector creates connection between business interfaces. There can be three types of connectors - the method invocation connector, the stream connector and the message connector. The following paragraphs will at first deal with the transformation of a general connector element from which every specific connector element inherits its provided and required ports. Then the transformations of particular connector types will be described.

3.5.1 Connector element

Description

A connector element is a building block of any connector. It provides a subset of four interfaces described in section 2.5. The interfaces are used by the mechanism which connects the elements among each other and with the delegation chains.

Transformations

Table 3.15: Simple component prototypes

SOFA 2 entity	Prototype name
Connector element	`conel`

Table 3.16: Provided ports by `conel`

Port name	Signature
`ElementLocalServer`	`org.objectweb.dsrg.connector.ElementLocalServer`
`ElementRemoteServer`	`org.objectweb.dsrg.connector.ElementRemoteServer`
`ElementRemoteClient`	`org.objectweb.dsrg.connector.ElementRemoteClient`
`ElementLocalClient`	`org.objectweb.dsrg.connector.ElementLocalClient`

The propriate port is present if and only if the connector element implements the corresponding interface.

3.5.2 Method invocation connector

Description

Method invocation connector (figure 2.4) contains four connector elements. Two of them are connector units (client unit, server unit) and two of them are responsible for propagating the business calls (local stub, local skeleton).

Transformations

Table 3.17: Simple component prototypes

SOFA 2 entity	Prototype name
Client unit	`conelClientUnit`
Server unit	`conelServerUnit`
Local stub	`conelLocalStub`
Local Skeleton	`conelLocalSkeleton`

Table 3.18: Provided ports by `conelLocalStub`

Port name	Signature
`call`	"business interface signature whose calls the connector element propagates"

Table 3.19: Required ports by `conelLocalStub`

Port name	Signature
`line`	"business interface signature whose calls the connector element propagates"

Table 3.20: Provided ports by `conelLocalSkeleton`

Port name	Signature
`line`	"business interface signature whose calls the connector element propagates"

Table 3.21: Required ports by `conelLocalSkeleton`

Port name	Signature
`call`	"business interface signature whose calls the connector element propagates"

Table 3.22: Connections

Requiring prototype	Required port	Providing prototype	Provided port
`conelLocalStub`	`line`	`conelLocalSkeleton`	`line`

3.5.3 Stream connector

Description

The stream connector (figure 2.5) contains only two connector elements. The streaming stub and the streaming skeleton connected through a socket.

Transformations

Table 3.23: Simple component prototypes

SOFA 2 entity	Prototype name
Streaming stub	`conelStreamingStub`
Streaming skeleton	`conelStreamingSkeleton`

Table 3.24: Provided ports by `conelStreamingStub`

Port name	Signature
`call`	"business interface signature whose calls the connector element propagates"

Table 3.25: Required ports by `conelStreamingSkeleton`

Port name	Signature
`call`	"business interface signature whose calls the connector element propagates"

The connection between the streaming stub and the streaming skeleton is established through the global connector manager. It asks the streaming skeleton through its `ElementRemoteSever` port for its location (IP address and port) which is then delivered to the streaming stub through its `ElementRemoteClient` interface. The business interface of the port `call` provides functions for retrieving the stream connection.

3.5.4 Message connector

Description

The message connector (figure 2.6) uses seven connector elements. Two of them are connector units (Sender Unit, Push Recipient) and the rest of them (Send adaptor, twice Send Receive, Distribution unit, Receive Adaptor) is responsible for propagating the message from the sender to the recipient. The message connector can connect interfaces which do not have the same signature. The connection can comprise a special interface for sending messages and a different one for receiving messages.

Transformations

Table 3.26: Simple component prototypes

SOFA 2 entity	**Prototype name**
Sender unit	`conelSenderUnit`
Push Recipient	`conelPushRecipient`
Sender adaptor	`conelSendAdaptor`
Send Receive	`conelSendReceive`
Distribution unit	`conelDistributionUnit`
Receive Adaptor	`conelReceiveAdaptor`

Table 3.27: Provided ports by `conelSendAdaptor`

Port name	**Signature**
`in`	"signature of the business interface for sending messages"

Table 3.28: Required ports by `conelSendAdaptor`

Port name	**Signature**
`out`	`org.objectweb.dsrg.connector.messaging.MessageSender`

Table 3.29: Provided ports by `conelSendReceive`

Port name	Signature
`send`	`org.objectweb.dsrg.connector.messaging.MessageSender`

Table 3.30: Required ports by `conelSendReceive`

Port name	Signature
`recv`	`org.objectweb.dsrg.connector.messaging.MessageReceiver`

Table 3.31: Provided ports by `conelReceiveAdaptor`

Port name	Signature
`in`	`org.objectweb.dsrg.connector.messaging.MessageReceiver`

Table 3.32: Required ports by `conelReceiveAdaptor`

Port name	Signature
`out`	"signature of the business interface for receiving messages"

Table 3.33: Connections

Requiring prototype	Required port	Providing prototype	Provided port
`conelSendAdaptor`	`out`	`conelSendReceive`	`send`
`conelSendReceive`	`recv`	`conelReceiveAdaptor`	`send`

The connection between the send receive components and the distribution unit is managed by the global connector manager which establishes the connection in the same way as in the case of the stream connection.

3.6 Composite components

The transformation of the composite component concept to a simple component architecture exploits the transformations mentioned in the previous sections 3.4 and 3.5.

The concept of composite components is realized through one outer delegation chain representing the outer business interface, one inner delegation chain representing the inner business interface and one connector connecting these two interfaces. As described in section 3.4 such interfaces are transformed into a simple component architecture constituted of a `delgChain`s representing the delegation chains and a set of `mc`s representing the microcom-

ponents in the delegation chains. The section 3.5 described transformations of connectors of various types. This forms three simple component architectures and the transformation of the composite components concept puts these transformations together forming one simple component architecture.

The connections between the three simple architectures are parallel to the connections of microcomponents with the connector elements. Let `mcLast` be the last `mc` in the `delgChain` which is the transformation of the outer delegation chain. Let `conelFirst` be the `conel` representing the connector element to which the business call is handed over from the outer delegation chain. Let `conelLast` be the `conel` representing the connector element which hands the business call over to the inner delegation chain. Let `mcFirst` be the first `mc` in the `delgChain` which is the transformation of the inner delegation chain.

The `mcLast` has the `delgRequired` required port having the same signature as the business interfaces. The same business interface signature has the port provided by the `conelFirst`. Connection between these ports forms connection between the two simple component architectures representing the outer delegation chain and the connector. In the same way can be connected the required port of the `conelLast` with the provided port of the `mcFirst` simple component both sharing the same business interface signature. These two connections put the three simple component architectures together.

3.7 Factory evolution pattern

The factory evolution pattern is used for creation of new components and it brings one specific microcomponent.

3.7.1 Factory interceptor

Description

Factory interceptor is the microcomponent which handles created components at runtime. It is inside the business component which invokes the creation.

Transformations

Table 3.34: Simple component prototypes

SOFA 2 entity	Prototype name
Factory interceptor	`mcFactoryInterceptor`

Table 3.35: Provided ports by `mcFactoryInterceptor`

Port name	Signature
`delgProvided`	"signature of the factory interface"

Table 3.36: Required ports by `mcFactoryInterceptor`

Port name	Signature
`delgRequired`	"signature of the factory interface"
`toComponent`	`org.objectweb.dsrg.sofa.microarchitecture.MIComponent`

3.8 Putting all in one

This chapter proposed transformation patterns on how to transform particular SOFA 2 concepts to a simple component architecture. This section gives a summary of these transformations describing how to apply them altogether when transforming the whole SOFA 2 component architecture.

Every primitive business component is transformed to one `core` having every required and provided port connected to the chain of `mc`s which mediate the communication with the outer world. These `mc`s are connected to a network of `conel`s which realize connection with other business components.

For a composite component there need to be transformed its connections between the outer business interfaces and the inner business interfaces. As describes section 3.6 every such connection is transformed to a simple architecture representing the outer delegation chain, connector and the inner delegation chain.

The control part of each business component is a network of interconnected `mc`s through their provided and required ports. Some of these `mc`s serve as the entry points of the business component's control interfaces.

The rest of the simple components (`delgChain`, `componentInstance`) are used for gathering logically related simple components and making them accessible to the environment.

Chapter 4

Proof of the unifying concept

The previous chapter brought a new point of view at all concepts that can be employed in a SOFA 2 component architecture. It introduced a set of transformations mapping them onto a lower level of abstraction trying to hide their mutual heterogenity. This chapter aims to bring the synthesis of the proposed transformations and prove that the unifying concept is correct and feasible.

For the sake of proving the concept, the following sections will examine several component architectures each of which employs one of the advanced concepts SOFA 2 component architecture can employ. Every architecture will be described from the business point of view and then modelled using a simple component architecture according to the proposed transformation patterns put forward in previous chapter.

All the models have been implemented in Java programming language using the OSGi [10] framework. Every simple component is represented by an OSGi bundle [11] (further on referred to as the component bundle) reflecting the black box nature of the simple component. The interface definitions the ports are associated with are also in separate OSGi bundles (further on referred to as the interface bundles). This makes the component bundles include the interface bundles if they want to communicate with other components through a port. This underlines that particular components can see other components only through these interfaces which is a fundamental characteristics of any component system.

The simple component models have been developed in Eclipse [13]. Each model is associated with one workspace made of OSGi projects. These projects represent the component and interface bundles together with subsidiary bundles offering auxiliary tools for the simple components management. Every component bundle contains an activator which instantiates the simple component it stands for and registers it as an OSGi service [12] in the provided bundle context. Each workspace contains a unique OSGi bundle, the launcher, which holds the information of connections between particular simple components and takes care of retrieving these components from the bundle context and connecting them properly. Each such connection represents the connection between two ports in the simple component architecture. Once the connections are established the model runs the architecture by launching the `core`s components.

The implementation of each simple component architecture is extracted from the real SOFA 2 implementation and strictly cut off from any of the subsidiary parts. Some of them were modified in order to depart them from the mesh of SOFA 2 instantiating managers.

4.1 Foreword

The first series of architectuers will be dedicated to connectors. The first modelled architecture will apart from all demonstrate full application of aspects creating a network of microcomponents around each of the business content. The control part is nearly the same for each of the business component which is why the rest of models will miss the microcomponents for the sake of keeping them clear and understandable. The models will include only those microcomponents which bring special features.

4.2 Template for model transformation description

The following sections will have identical structure as they will describe transformations of particular component architectures. Each section will inform about the transformation pattern it focuses on along with brief description of the component architecture on which the transformations will be demonstrated. The description will be accompanied with two figures. The first one will show the business point of view at the architecture and the second one will reveal its internal runtime parts. Every logical part will be given a name which will be referenced from the following tables showing all transformations applied on the component architecture elements. The tables will also contain entities not shown in the figures. Such entities will be those which only serve as containers of other entities (business components, delegation chains and connector units). They will be given a unique description. The tables will also show connections between the ports of transformed simple components. The result of the transformations will form a simple component architecture representing the component architecture on the lower level of abstraction.

4.3 Method invocation connector

The transformations will kick off with an example focusing on the method invocation connector. The component architecture designed to demonstrate the method invocation connector consists of two components (figure 4.1) connected through one `log` interface using the method invocation connector. The interface contains one function having one `String` parameter. The tester calls the function with parameter `"Hello world"` and the logger prints the message in the console. The architecture implementation can be found in the enclosed CD in the workspace `LogDemo`.

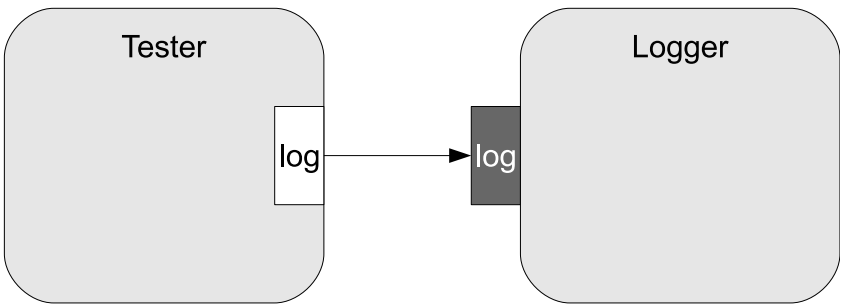


Figure 4.1: Method invocation architecture

Before proceeding forward, the following list will introduce aspects applied on both components. Each aspect will give brief description of control interfaces and microcomponents it supplies the business components with. The microcomponents will be given names corresponding to those given to the microcomponents' graphical reproduction depicted in figures 4.2 and 4.3.

- `Component`. The `Component` aspect supplies the component with one control interface and one microcomponent (`control`) representing the delegation chain of that interface. The microcomponent provides the outer world with access to all business component's interfaces and to the business content.

- `InComponent`. The `InComponent` aspect has been described in subsection 2.4.1. It supplies the component with an ability of tracing threads calling business interfaces. It supplies the architecture with microcomponents `interceptor`, `base` and `impl`.

- `Lifecycle`. The `Lifecycle` aspect supplies the component with one control interface and two microcomponents. The first (`impl2`) represents the delegation chain of the control interface and is connected to the second microcomponent (`base2`) which is a standalone unit responsible for managing the component's lifecycle. The aspect also supplies all delegation chains of provided business interfaces with a microcomponent also keeping track of threads calling the business interface.

The component architecture consists of two components - the tester and the logger. The tester component contains one business `content` and four delegation chains. Three of them represent the three control interfaces supplied by aspects and the fourth represents the business required interface. The fourth delegation chain is made of one microcomponent - the `interceptor`.

The logger component also contains one business `content` and four delegation chains. Their structure is the same as in the tester component except for the delegation chain for the business provided interface which has one more microcomponent, the `interceptor2`.

These components are connected through the method invocation connector consisting of one client connector unit, one sever connector unit and two connector elements - the `local stub` and the `local skeleton`. These connector elements are connected to the microcomponents representing the business interfaces delegation chains.

The following figures (4.2, 4.3) show the logical entities of the discussed component architecture. They do not show the logical entities which only serve as containers of other logical entities. In other words they show the business component contents, all microcomponents representing the control parts and the method invocation connector elements.

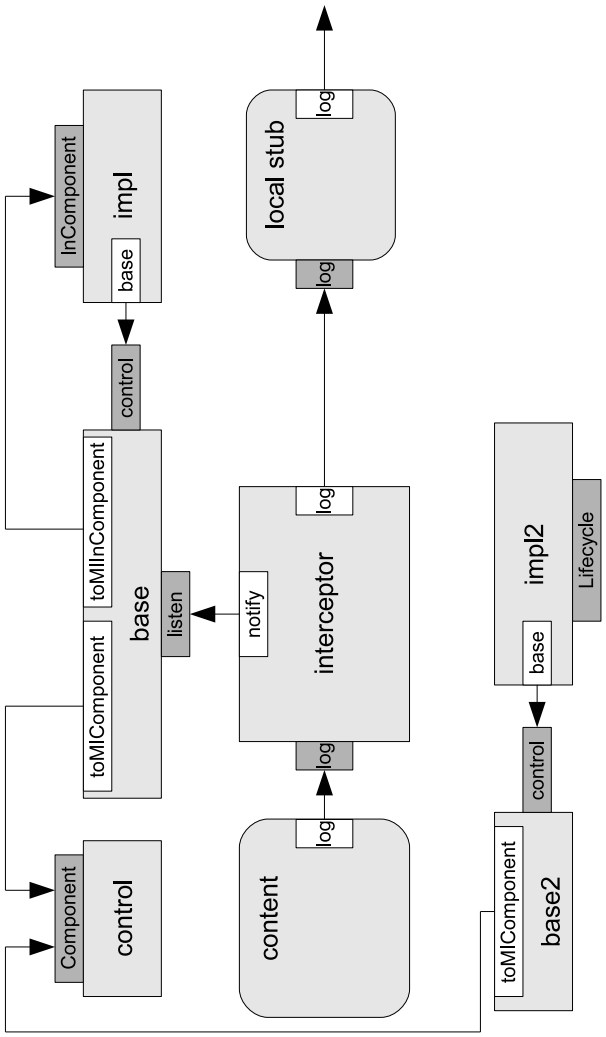


Figure 4.2: Method invocation architecture (tester part)

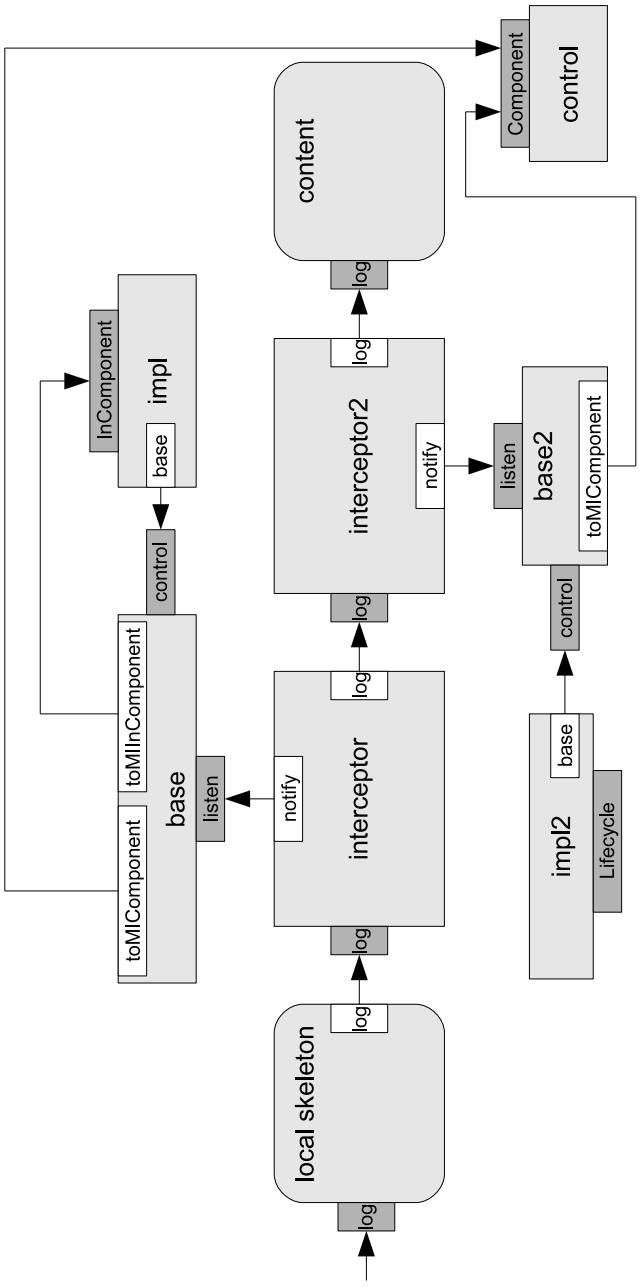


Figure 4.3: Method invocation architecture (logger part)

In order to transform this component architecture into the simple component architecture, it is necessary to employ transformation patterns on these concepts:

- basic concept for transformation of business components and their contents
- control part concept for transformation of delegation chains and microcomponents
- method invocation connector concept for transformation of the connection between the two components

The following tables (4.1, 4.2) show the transformations of all logical entities in the discussed component architecture.

Table 4.1: Transformation table for the tester part

SOFA 2 entity	**Prototype name**
`content`	`core`
`control`	`mc`
`base`	`mc`
`impl`	`mc`
`base2`	`mc`
`impl2`	`mc`
`interceptor`	`mc`
`client unit` connector unit	`conelClientUnit`
`local stub`	`conelLocalStub`
Component control interface	`delgChain`
Lifecycle control interface	`delgChain`
InComponent control interface	`delgChain`
Required log interface	`delgChain`
Tester component	`componentInstance`

Table 4.2: Transformation table for the logger part

SOFA 2 entity	**Prototype name**
`content`	`core`
`control`	`mc`
`base`	`mc`
`impl`	`mc`
`base2`	`mc`
`impl2`	`mc`
`interceptor`	`mc`
`interceptor2`	`mc`
`server unit` connector unit	`conelServerUnit`
`local skeleton`	`conelLocalSkeleton`
Component control interface	`delgChain`
Lifecycle control interface	`delgChain`
InComponent control interface	`delgChain`
Provided log interface	`delgChain`
Logger component	`componentInstance`

The following tables (4.3, 4.4) show connections between particular simple components. Their names are inherited from the logical entities they were transformed from. The tables at the same time reveal the list of provided and required ports of particular simple components.

Table 4.3: Connections in the tester part

Requiring prototype	Required port	Providing prototype	Provided port
content	log	interceptor	delgProvided
interceptor	notify	base	listen
base	toMIComponent	control	Component
base	toMIInComponent	impl	InComponent
impl	base	base	control
interceptor	delgRequired	local stub	call
local stub	line	local skeleton	line
base2	toMIComponent	control	Component
impl2	base	base2	control

Table 4.4: Connections in the logger part

Requiring prototype	Required port	Providing prototype	Provided port
local stub	line	local skeleton	line
local skeleton	call	interceptor	delgProvided
interceptor	notify	base	listen
base	toMIComponent	control	Component
base	toMIInComponent	impl	InComponent
impl	base	base	control
interceptor	delgRequired	interceptor2	delgProvided
interceptor2	delgRequired	content	log
interceptor2	notify	base2	listen
base2	toMIComponent	control	Component
impl2	base	base2	control

There is one simple component having one provided port which has not been covered in previous tables. It is the `impl2` component and the port name is `Lifecycle`. The interfaces the components' ports are associated with are determined by the transformation patterns described in previous chapter.

4.4 Stream connector

This section will focus on the stream connector. The architecture demonstrating the stream connector is depicted in figure 4.4. It consists of two components, the `Stream consumer` and the `Stream producer`. They are connected through two interfaces, the `stream provider` and the `media server`. The connection through the stream provider interface is realized by a stream connector. The connection through the media server interface is realized by a method invocation connector. The media server interface contains functions for retrieving list of provided resources. The interface stream provider is designed to provide a resource based on its id returned from the media server interface. The resource is provided as a stream of data. The architecture implementation can be found in the enclosed CD in the workspace `StreamDemo`.

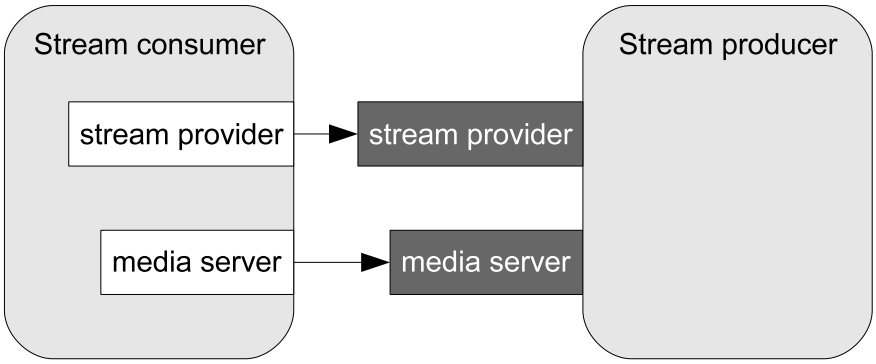


Figure 4.4: Stream connector architecture

The following description of logical entites constituting the stream connector architecture will no longer mention the control part as it was comprehensively covered in previous section describing transformation of the method invocation architecture.

The stream consumer component consists of one business content and two delegation chains representing the business required interfaces the `media server` and the `stream provider`. They are both made of a single microcomponent, the `interceptor` and the `interceptor stream` respectively both delivered by the `InComponent` aspect. The stream consumer is depicted in figure 4.5.

Figure 4.6 shows internals of the stream provider component. It consists of one business content and two delegation chains representing the business provided interfaces, the `media server` and the `stream provider`. They are both made of two microcomponents. The one representing the `media server` interface is made of the microcomponents `interceptor` and `interceptor2`. The one representing the `stream provider` interface is made of the microcomponents `interceptor stream` and `interceptor stream2`. The microcomponents are result of application of the `InComponent` and the `Lifecycle` aspect respectively.

The `media server` interface connector is the method invocation connector consisting of one client connector unit, one sever connector unit and two connector elements - the `local stub` and the `local skeleton`. These connector elements are connected to the microcomponents representing the business interfaces delegation chains and between each other they are connected through a direct reference. The `stream provider` interface connector is the stream connector consisting of two connector elements - the `streaming stub` and the `streaming`

`skeleton`. These connector elements are connected to the microcomponents representing the business interfaces delegation chains. The connector elements are connected through a socket.

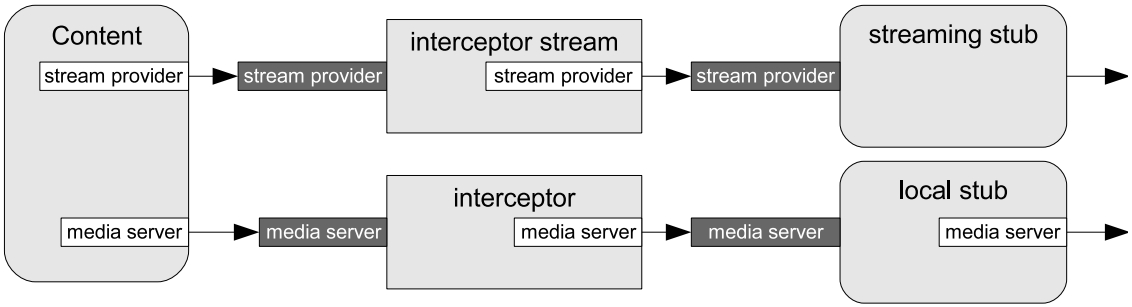


Figure 4.5: Stream connector architecture (stream consumer part)

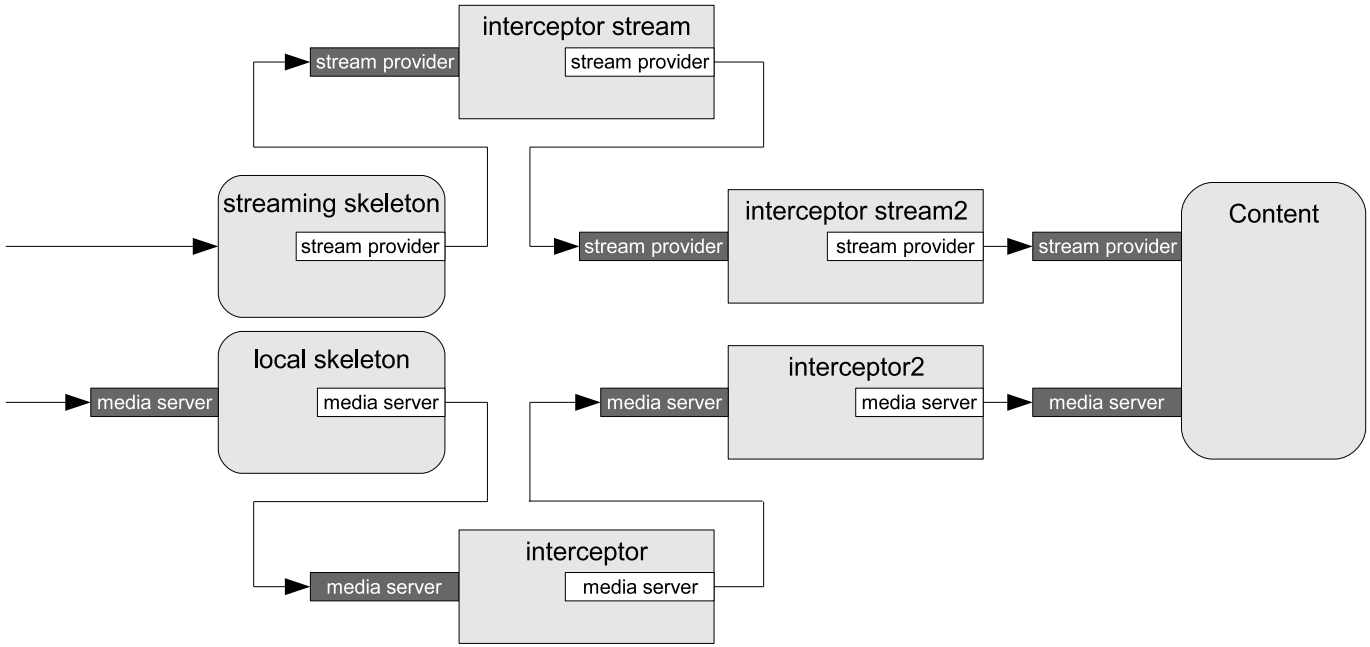


Figure 4.6: Stream connector architecture (stream producer part)

In order to transform this component architecture into the simple component architecture, it is necessary to employ transformation patterns on these concepts:

- basic concept for transformation of business components and their contents
- control part concept for transformation of delegation chains and microcomponents
- method invocation connector concept for transformation of the connection through the `media server` interface
- stream connector concept for transformation of the connection through the `stream provider` interface

The following tables (4.5, 4.6) show the transformations of all logical entities in the discussed component architecture.

Table 4.5: Transformation table for the stream consumer part

SOFA 2 entity	**Prototype name**
`content`	`core`
`interceptor`	`mc`
`interceptor stream`	`mc`
`local stub`	`conelLocalStub`
`streaming stub`	`conelStreamingStub`
`client unit`	`conelClientUnit`
Required media server interface	`delgChain`
Required stream provider interface	`delgChain`
Stream consumer component	`componentInstance`

Table 4.6: Transformation table for the stream producer part

SOFA 2 entity	Prototype name
content	core
interceptor	mc
interceptor2	mc
interceptor stream	mc
interceptor stream2	mc
local skeleton	conelLocalSkeleton
streaming skeleton	conelStreamingSkeleton
server unit	conelServerUnit
Provided media server interface	delgChain
Provided stream provider interface	delgChain
Stream producer component	componentInstance

The following tables (4.7, 4.8) show connections between particular simple components. Their names are inherited from the logical entities they were transformed from. The tables at the same time reveal the list of provided and required ports of particular simple components.

Table 4.7: Connections in the stream consumer part

Requiring prototype	Required port	Providing prototype	Provided port
content	stream provider	interceptor stream	delgProvided
content	media server	interceptor	delgProvided
interceptor stream	delgRequired	streaming stub	call
interceptor	delgRequired	local skeleton	call
local stub	line	local skeleton	line

Table 4.8: Connections in the stream producer part

Requiring prototype	Required port	Providing prototype	Provided port
local stub	line	local skeleton	line
streaming skeleton	call	interceptor stream	delgRequired
local skeleton	call	interceptor	delgProvided
interceptor stream	delgRequired	interceptor stream2	delgProvided
interceptor	delgRequired	interceptor2	delgProvided
interceptor stream2	delgRequired	content	stream provider
interceptor2	delgRequired	content	media server

4.5 Message connector

This section will focus on the message connector. The architecture demonstrating the message connector is depicted in figure 4.7. It consists of two components, the `Producer` and the `Consumer`. They are connected through a message connector. The `Producer` component generates messages and sends them via the `news publisher` business interface. The message connector distributes the message to the subscriber through its `news subscriber` business interface. The architecture implementation can be found in the enclosed CD in the workspace `MessageDemo`.

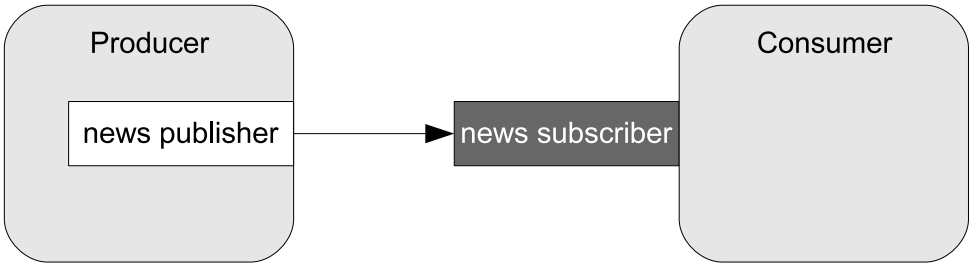


Figure 4.7: Message connector architecture

Both components are again represented by their content and one delegation chain reflecting the business interface. The microcomponents constituting the delegation chains have been delivered by the `InComponent` and the `Lifecycle` aspect. The delegation chain in the message producer part consists of one microcomponent, the `interceptor`. The delegation chain in the message consumer part consists of two microcomponents, the `interceptor` and the `interceptor2`. The connector on both sides of communication consists of the message transformation part (`adaptor`s and the `send receive` units) which is connected to the microcomponents representing the business interfaces. It also translates the message to/from the serialized form and sends/receives it to/from the distribution unit to which it is connected through a socket. All entities are revealed in figures 4.8 and 4.9.

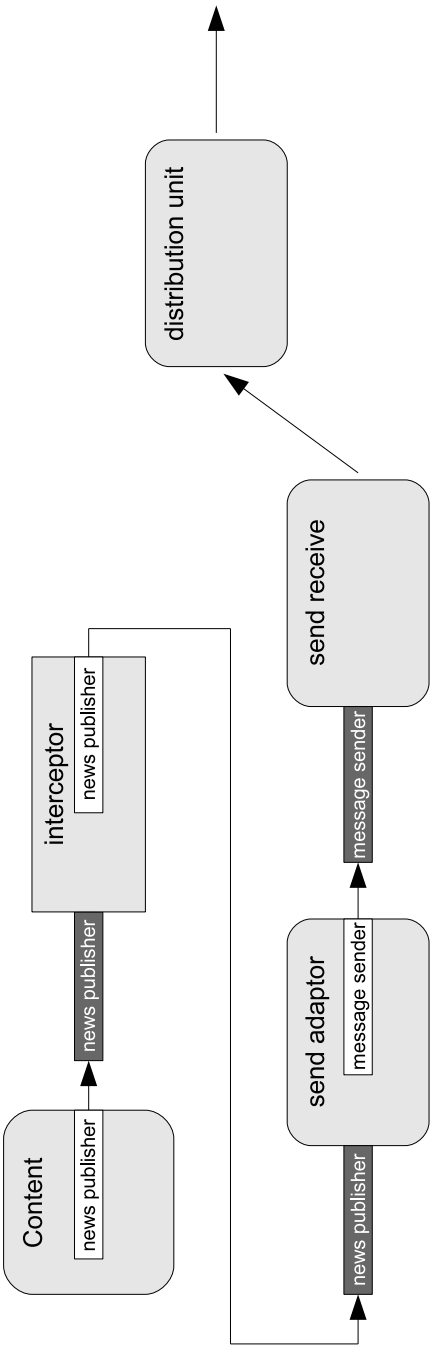


Figure 4.8: Message connector architecture (message producer part)

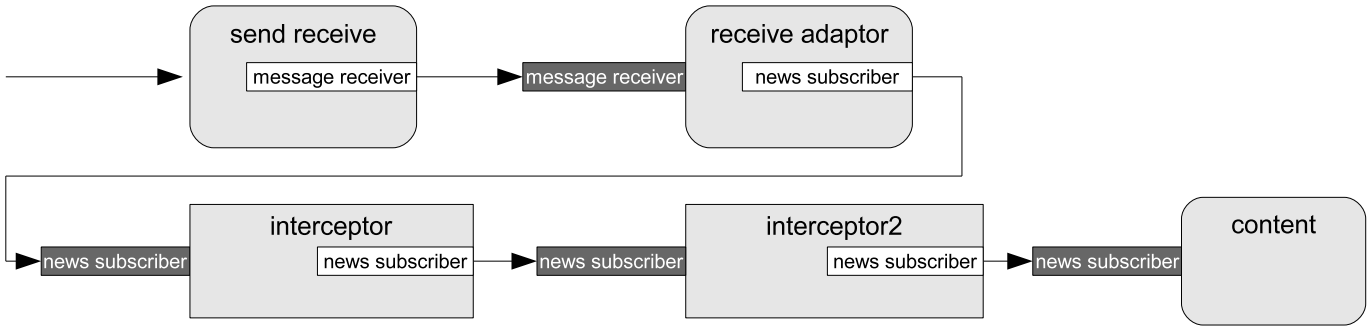


Figure 4.9: Message connector architecture (message consumer part)

In order to transform this component architecture into the simple component architecture, it is necessary to employ transformation patterns on these concepts:

- basic concept for transformation of business components and their contents
- control part concept for transformation of delegation chains and microcomponents
- message connector concept for transformation of the connection between the producer and the consumer component

The following tables (4.9, 4.10) show the transformations of all logical entities in the discussed component architecture.

Table 4.9: Transformation table for the message producer part

SOFA 2 entity	**Prototype name**
`content`	`core`
`interceptor`	`mc`
`Sender unit` connector unit	`conelSenderUnit`
`send adaptor`	`conelSendAdaptor`
`send receive`	`conelSendReceive`
`distribution unit`	`conelDistributionUnit`
Required news publisher interface	`delgChain`
Producer component	`componentInstance`

Table 4.10: Transformation table for the message consumer part

SOFA 2 entity	**Prototype name**
`Push Recipient` connector unit	`conelPushRecipient`
`send receive`	`conelSendReceive`
`receive adaptor`	`conelReceiveAdaptor`
`interceptor`	`mc`
`interceptor2`	`mc`
`content`	`core`
Required news subscriber interface	`delgChain`
Consumer component	`componentInstance`

The following tables (4.11, 4.12) show connections between particular simple components. Their names are inherited from the logical entities they were transformed from. The tables at the same time reveal the list of provided and required ports of particular simple components.

Table 4.11: Connections in the message producer part

Requiring prototype	Required port	Providing prototype	Provided port
content	news publisher	interceptor	delgProvided
interceptor	delgRequired	send adaptor	in
send adaptor	out	send receive	send

Table 4.12: Connections in the message consumer part

Requiring prototype	Required port	Providing prototype	Provided port
send receive	recv	receive adaptor	in
receive adaptor	out	interceptor	delgProvided
interceptor	delgRequired	interceptor2	delgProvided
interceptor2	delgRequired	content	news subscriber

4.6 Composite components

This section will focus on the composite components concept. The architecture demonstrating the concept is depicted in figure 4.10. It contains two components, the `Tester` and the `Composite Logger`. The `Composite Logger` is a composite component and contains one subcomponent, the `Logger`. The `Tester` and the `Composite Logger` are connected by a method invocation connector connecting their `log` interfaces. The `Composite Logger` delegates the `log` interface to its subcomponent providing the same interface `log`. The architecture implementation can be found in the enclosed CD in the workspace `CompositeLogDemo`.

The `Tester` component consists only of two SOFA 2 entities. The content and one microcomponent standing for the delegation chain of the required interface `log`.

The `Composite Logger` component covers the delegation chain of the provided business interface `log` and the connector connecting the interface with the subcomponent `Logger`. The delegation chain is made of one microcomponent, the `composite interceptor2`, delivered by the `Lifecycle` aspect. The connector is constituted of two connector units and their subelements - the `composite local stub` and the `composite local skeleton`.

The `Logger` component is represented by the content and two microcomponents, the `interceptor` and the `interceptor2`, standing for the delegation chain of the provided interface `log`.

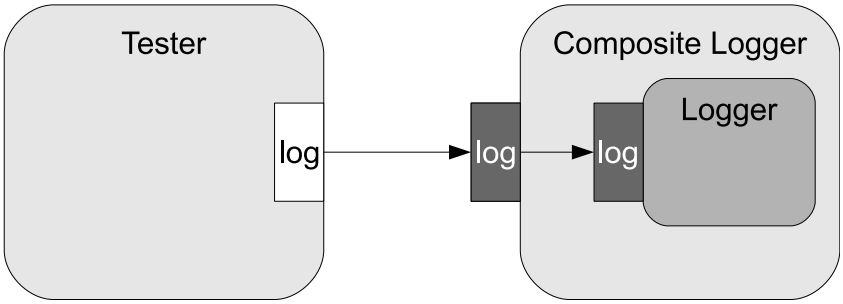


Figure 4.10: Composite component architecture

The connector connecting the `Tester` and the `Composite Logger` component consists of the connector units and their subelements - the `local stub` and the `local skeleton`. All entities are revealed in figures 4.11 and 4.12.

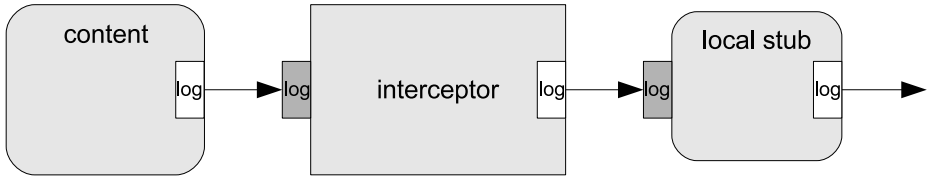


Figure 4.11: Composite component architecture (tester part)

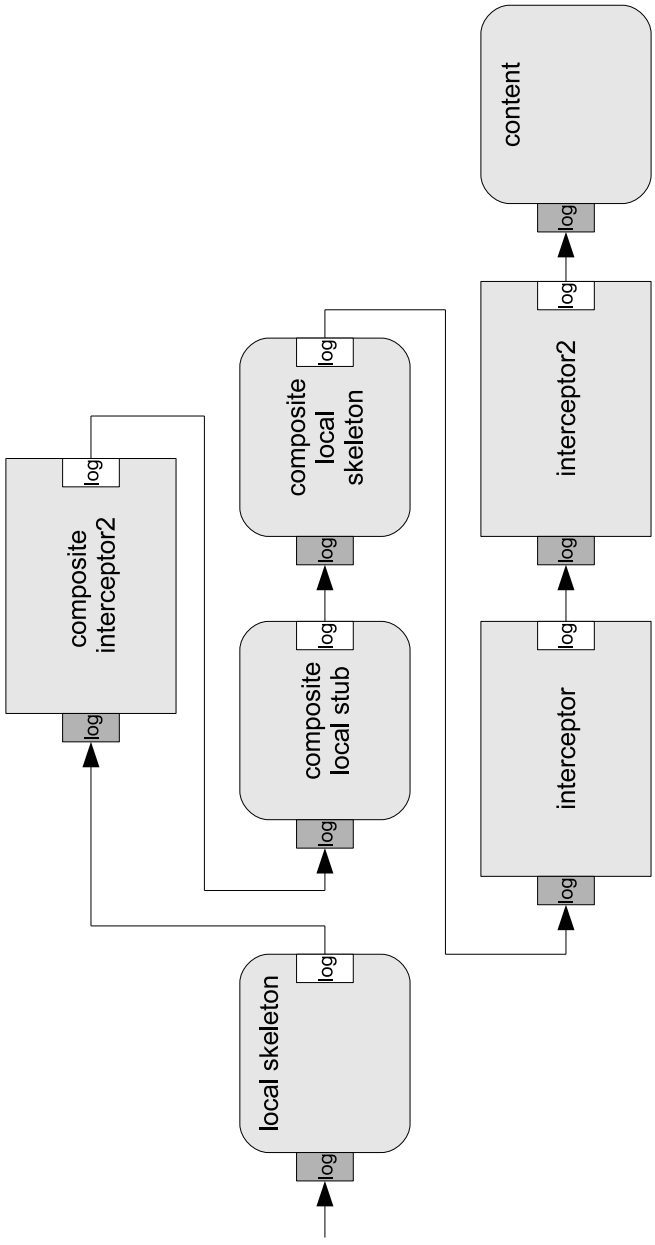


Figure 4.12: Composite component architecture (logger part)

In order to transform this component architecture into the simple component architecture, it is necessary to employ transformation patterns on these concepts:

- basic concept for transformation of business components and their contents
- control part concept for transformation of delegation chains and microcomponents
- method invocation connector concept for transformation of the connection between the two components and for the transformation of the connection between the outer and inner interface of the `Composite Logger` component

The following tables (4.13, 4.14, 4.15) show the transformations of all logical entities in the discussed component architecture.

Table 4.13: Transformation table for the tester part

SOFA 2 entity	**Prototype name**
`content`	`core`
`interceptor`	`mc`
`client unit` connector unit	`conelClientUnit`
`local stub`	`conelLocalStub`
Required log interface	`delgChain`
Tester component	`componentInstance`

Table 4.14: Transformation table for the composite logger part

SOFA 2 entity	**Prototype name**
`server unit` connector unit	`conelServerUnit`
`local skeleton`	`conelLocalSkeleton`
`composite interceptor2`	`mc`
`composite client unit` connector unit	`conelClientUnit`
`composite local stub`	`conelLocalStub`
`composite server unit` connector unit	`conelServerUnit`
`composite local skeleton`	`conelLocalSkeleton`
Composite Logger provided log interface	`delgChain`
Composite Logger component	`componentInstance`

Table 4.15: Transformation table for the logger part

SOFA 2 entity	Prototype name
interceptor	mc
interceptor2	mc
content	core
Provided log interface	delgChain
Logger component	componentInstance

The following tables (4.16, 4.17) show connections between particular simple components. Their names are inherited from the logical entities they were transformed from. The tables at the same time reveal the list of provided and required ports of particular simple components.

Table 4.16: Connections in the tester part

Requiring prototype	Required port	Providing prototype	Provided port
content	log	interceptor	delgProvided
interceptor	delgRequired	local stub	call
local stub	line	local skeleton	line

Table 4.17: Connections in the composite logger part

Requiring prototype	Required port	Providing prototype	Provided port
local stub	line	local skeleton	line
local skeleton	call	composite interceptor2	delgProvided
composite interceptor2	delgRequired	composite local stub	call
composite local stub	line	composite local skeleton	line
composite local skeleton	call	interceptor	delgProvided
interceptor	delgRequired	interceptor2	delgProvided
interceptor2	delgRequired	content	log

4.7 Factory pattern

The final section will be dedicated to factory evolution pattern. The component architecture designed to demonstrate its transformation is depicted in figure 4.13. It is composed of two components, the `Tester` and the `Factory`. They are connected through their `factory` interface by means of a method invocation connector. The `Factory` component creates new components, the `Logger`s, whose business model shows figure 4.14. Upon invocation of the factory function in the `factory` interface the `Factory` component creates new instance of the `Logger` component and returns reference on one of its provided interfaces chosen according to annotations of the `factory` interface. The architecture implementation can be found in the enclosed CD in the workspace `FactoryDemo`.

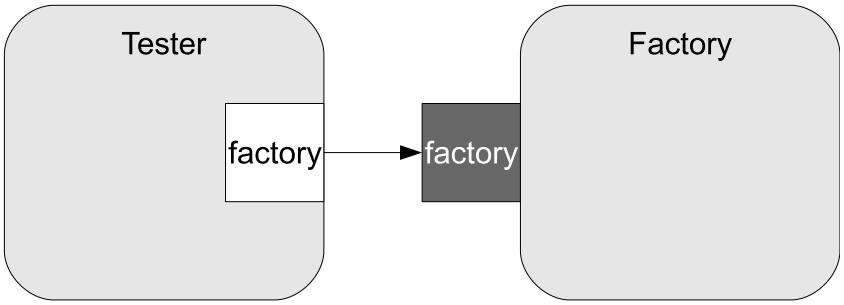


Figure 4.13: Factory evolution pattern architecture

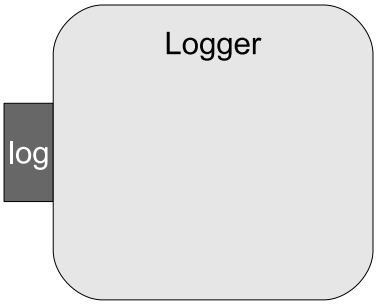


Figure 4.14: Logger component

The `Tester` component is composed of one content and one delegation chain representing the required `factory` interface. It is made of two microcomponents - the `factory interceptor` and the `interceptor`. The `factory interceptor` takes care of injecting new `Logger` components in the required collection inside the content. Figure 4.15 depicts the situation with one `Logger` component already injected in the collection. The `factory interceptor` is delivered by the `FactoryPattern` aspect which inserts the `factory interceptor` microcomponent in the required factory delegation chain. It acquires access to the component's content through the `Component` control interface. The `interceptor` microcomponent is delivered by the `InComponent` aspect.

The `Factory` component is constituted of one content and one delegation chain representing the provided `factory` interface. It is formed by two microcomponents - the `interceptor` and the `interceptor2` delivered by the `InComponent` and the `Lifecycle` aspects respectively.

The connector connecting the `Tester` and the `Factory` component is made of two connector units both having one subelement - the `local stub` and the `local skeleton`. All entities are revealed in figures 4.15 and 4.16.

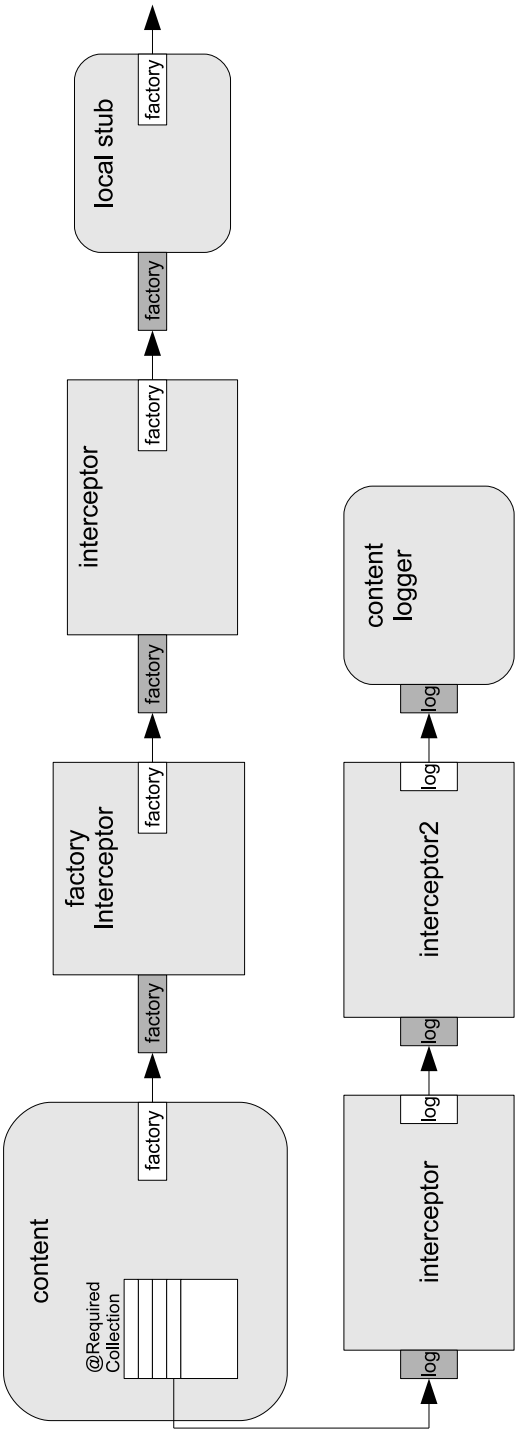


Figure 4.15: Factory evolution pattern architecture (tester part)

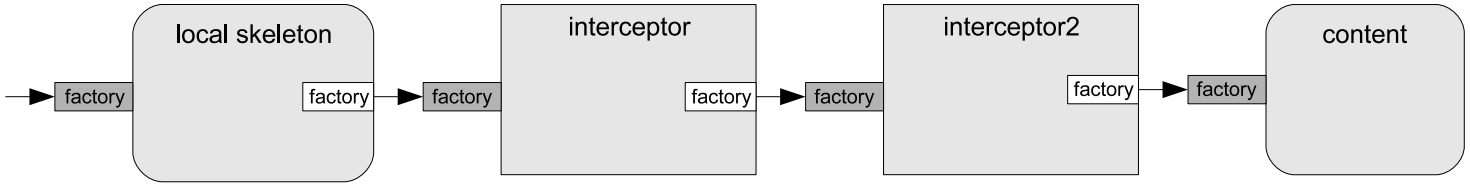


Figure 4.16: Factory evolution pattern architecture (factory part)

In order to transform this component architecture into the simple component architecture, it is necessary to employ transformation patterns on these concepts:

- basic concept for transformation of business components and their contents
- control part concept for transformation of delegation chains and microcomponents
- method invocation connector concept for transformation of the connection between the two components
- factory evolution pattern concept for transformation of the factory interceptor

The following tables (4.18, 4.19) show the transformations of all logical entities in the discussed component architecture. They do not include the injected `Logger` component.

Table 4.18: Transformation table for the tester part

SOFA 2 entity	**Prototype name**
`content`	`core`
`factory interceptor`	`mc`
`interceptor`	`mc`
`client unit` connector unit	`conelClientUnit`
`local stub`	`conelLocalStub`
Required factory interface	`delgChain`
Tester component	`componentInstance`

Table 4.19: Transformation table for the factory part

SOFA 2 entity	**Prototype name**
`server unit` connector unit	`conelServerUnit`
`local skeleton`	`conelLocalSkeleton`
`interceptor`	`mc`
`interceptor2`	`mc`
`content`	`core`
Provided log interface	`delgChain`
Logger component	`componentInstance`

The following tables (4.20, 4.21) show connections between particular simple components. Their names are inherited from the logical entities they were transformed from. The tables at the same time reveal the list of provided and required ports of particular simple components. They do not include the injected `Logger` component.

Table 4.20: Connections in the tester part

Requiring prototype	**Required port**	**Providing prototype**	**Provided port**
`content`	`factory`	`factory interceptor`	`delgProvided`
`factory interceptor`	`delgRequired`	`interceptor`	`delgProvided`
`interceptor`	`delgRequired`	`local stub`	`call`
`local stub`	`line`	`local skeleton`	`line`

Table 4.21: Connections in the factory part

Requiring prototype	**Required port**	**Providing prototype**	**Provided port**
`local stub`	`line`	`local skeleton`	`line`
`local skeleton`	`call`	`interceptor`	`delgProvided`
`interceptor`	`delgRequired`	`interceptor2`	`delgProvided`
`interceptor2`	`delgRequired`	`content`	`factory`

Chapter 5

Related work

The issue of transformation of a sophisticated model into a simple flat model has been subject of interest to OMG[1] deployment and configuration (OMG D&C) [14] project. This chapter will give a quick overview of the project and will give a brief comparison between its approach to the solution of this issue and the approach presented in this work.

OMG D&C is an attempt to define generic support for deployment of component-based applications. Its aim is to introduce a unified way for their deployment. For this sake it defines several abstraction levels used in the deployment process.

The first level of abstraction is the *component data model* which represents a logical point of view at the application model. It considers a component to be either a monolithic bunch of code or as an assembly of other components which introduces a recursive definition of a component. A building block of the component data model is the component package representing a reusable work product. A component package is a set of metadata and compiled code modules that contains implementations of a component interface. The implementations in a package can be a mix of monolithic and assembly implementations. Assemblies can consist of subcomponents whose implementations are inside the same package of software, or they can reference component packages that must exist in the environment outside the package containing the assembly. OMG D&C also supports heterogenous systems and for one component there can be more than one implementation (one for Macintosh, Linux or Windows).

Taking into account the complexity of the component data model and the potential target environment where the components would be deployed, it is necessary to introduce the *execution data model*, another abstraction level, which decides where to deploy each component and which one of its implementations should be selected for deployment such that it would best fit the target environment. It needs to resolve all recursively defined components, thus make the hierarchical component model flat and also needs to resolve references on components outside the packages they are referenced from. The execution data model is a simplified point of view at the component data model. The component data model is transformed to the execution data model before the application needs to be executed.

The approach to transformation presented in this work and the approach presented by the OMG D&C differ in great extent. Their nature is mainly driven by the purpose of the transformation. The purpose of the transformation of a SOFA 2 application employing heterogenous concepts is driven by the need of having a transparent and modular application

[1] www.omg.org

model unified by a simple metamodel. It constrasts to OMG D&C where the transformation of the component data model to the execution data model is driven by the business logic of the application. In the stage before the execution the deployment process does not need to view the application model in a sophisticated way and only needs to know several basic information which is why the transformation is performed. The two simplified models then differ in one thing. The simplified model of the SOFA 2 component architecture keeps the power of the model it was transformed from. It can still employ any of the advanced concepts as its logical parts represent the concepts themselves. The execution data model cannot use referencing components any more, it does not know the meaning of the term component assemblies and the configurations of component packages cannot be inherited.

Chapter 6

Conclusion and Future development

The thesis has presented the way how to capture heterogenous concepts found in SOFA 2 in one unifying concept. The concept brings new point of view at the component architecture of an arbitrary application. It views the architecture as a set of simple components without any advanced concept engaged. The simple components behave like a black box, thus it is not possible to distinguish which simple components represent the advanced concepts and the mutual heterogenity between the concepts is filtered.

The concept makes the application architecture transparent and much more flexible. It is possible to address any of its simple components and handle it in custom way. As the simple components communicate through ports associated with the same interface, it is possible to reshuffle them and customize their interconnections. Some of the simple components could be then completely removed from the result architecture which is the key achievement leading to its simplification. Let us for example consider a target environment consisting only of one deployment dock where the whole application should be deployed. It is obvious that the matter of seamless distribution is irrelavant. If we take closer look at the method invocation connector, it becomes clear that the local stubs and skeletons are no longer needed and that the connection can be made directly between the microcomponents forming the business delegation chains. Another subject of simplification can be the concept of composite components. The connection between the outer interface and the inner interface si realized through microcomponents of the outer interface, connector and the microcomponents of the inner interface. Their respective simple components are all connected through ports sharing the same interface. The connection could be shortened by the outer interface and the connector. The situation is parallel to transformations in OMG D&C where a bunch of assemblies and monolithic implementations is resolved to one flat bunch consisting only of monolithic implementations. In the same way the connections between the composite components and their subcomponents could be recursively resolved which could relieve the result architecture from redundant simple components.

The future development should focus on an automated transformation of a component architecture to its simple component architecture equivalent. It should be able to identify all concepts involved in an arbitrary application, transform the conceptual parts individually and then synthesize them such that they would reflect the former component architecture. Once the respective simple component architecture is formed, another tool should be employed which would optimize the architecture for the target environment where the application would be intended to be deployed. The output of this tool should be a new deployment plan saying what simple components should be employed in the application, where they should

be deployed and how they should be connected with the other simple components. This tool should also give a dependency graph of the simple components on other libraries they need in order to run correctly. For example connectors could need RMI or ActiveMQ [1] libraries. In order not to attach each library to each simple component, the libraries should be globally available such that the components could share them. As soon as the new deployment plan is generated the deployment process may begin. The role of the subsidiary parts would remain the same. Only the deployment dock would be instructed not to instantiate the business components but the simple components. The deployment dock would also connect the components according to the new deployment plan but only in the scope of components it would contain. The connections between components deployed in different docks would manage the global connector manager.

Bibliography

[1] Active MQ messaging provider - http://activemq.apache.org/

[Szyperski] Szyperski, C.: Component Software: Beyond Object-Oriented Programming, 2nd edition, Addison-Wesley, Jan 2002

[2] Tomáš Bureš, Petr Hnětynka: Advanced Features of Hierarchical Component Models

[3] Bures, T., Hnetynka, P., Plasil, F.: SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model, Proc. of SERA 2006, Seattle, USA, IEEE CS, Aug 2006

[4] N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Transactions on Software Engineering, January 2000.

[5] Magee, J.; Kramer, J.: Dynamic structure in software architectures, Proceedings of FSE'4, San Francisco, USA, Oct 1996

[6] R. Allen. "A Formal Approach to Software Architecture." Ph.D. Thesis, Carnegie Mellon University, CMU Technical Report CMUCS-97-144, May 1997 Cambridge University Press, 2000

[7] Enterprise Java Beans specification, version 3.0. Sun Microsystems, May 2006

[8] The Koala Component Model for Consumer Electronics Software, van Ommering, Frank van der Linden, Jeff Kramer, Jeff Magee, Computer, March 2000, p78-85.

[9] Remote Method Invocation (RMI) http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp

[10] OSGi Service Platform Core Specification, The OSGi Alliance Release 4, Version 4.1 April 2007

[11] OSGi bundle. http://en.wikipedia.org/wiki/OSGi#Bundles

[12] OSGi service. http://www.knopflerfish.org/osgi_service_tutorial.html

[13] Eclipse IDE, http://www.eclipse.org/

[14] Object Management Group: Deployment and Configuration of Component-based Distributed Applications Specification, Version 4, OMG document formal/06-04-02

Appendix A

Contents of the enclosed CD

The root of the enclosed CD contains this file structure:

- `workspaces` is the directory containing the workspaces of component applications discussed in chapter 4
- `workspaces/LogDemo` is the workspace demonstrating the method invocation architecture
- `workspaces/StreamDemo` is the workspace demonstrating the stream connector architecture
- `workspaces/MessageDemo` is the workspace demonstrating the message connector architecture
- `workspaces/CompositeLogDemo` is the workspace demonstrating the composite component architecture
- `workspaces/FactoryDemo` is the workspace demonstrating the factory evolution pattern architecture
- `doc` is the directory containing generated javadoc for particular projects
- `readme.txt` is a plaintext giving further information about the workspaces and it guides the user how to launch the applications represented by the workspaces
- `thesis.pdf` is a copy of this thesis